Solutions to the Computer Science Problems

Problem 1

(a) Why your program always produces a correct solution

My solution uses dynamic programming (dp) to store the maximum number of primes at each index of the input string.

The solution has three important parts:

- **The stored state**
  The maximum number of prime factors in the partition of digits up to $digit[i]$
- **The state transition**
  For each state, i, loop through all the previous states, $dp[j_x]$. Find $dp[i]$ by finding the maximum sum of $dp[j_x]$ and the number of primes factors in the substring from i to j_x.

$$dp[i] = \max(\text{number of prime factors}[j_x + 1 : i] + dp[j_x]) \text{ for } x \in [0, i-1]$$

- **The base case**
  The number of prime factors for the first 0 digits in the given string is 0.

$$dp[0] = 0$$

One can prove the correctness of this algorithm by using the exchange argument method, which compares the algorithm's output to a globally optimal solution that produces the maximum number of primes.

If one partitions the first i digits of the string with a locally non-optimal solution, one can prove that this will not produce a globally optimal solution. Changing the partition of the first i digits to be locally-optimal can only increase the number of primes in the partition. This consequently increases the total number of primes over the entire input string. As a result, any partition of digits must be locally optimal to maximize the total number of primes.

Thus, through constructing the dp algorithm to only build states from other locally-optimal solutions, the algorithm is globally optimal and we have proved the correctness of the algorithm.

(b) Worst Case Efficiency

The efficiency of my dp solution is $O(|n|^2 \cdot \sqrt[4]{n} \cdot n^{p_n})$ where $|n|$ represents the length of the string input. The $O(|n|^2)$ term is due to an $O(|n|)$ nested loop within another $O(|n|)$ loop. I am calling a combination of trial division and Pollard Rho's algorithm within these two loops to calculate the number of primes in the factorization of a number in $O(\sqrt[4]{n})$. $O(\sqrt[4]{n})$ is the worst case heuristic efficiency of the Pollard Rho algorithm. The last factor n^{p_n} is formed by running the worst case of the Pocklington deterministic algorithm where p_n is the number of primes of n. However, this factor is unrealistic for any given input.

The algorithm is extremely efficient for numbers with many small prime factors and less efficient for numbers with large prime factors.

(c) Average Case Efficiency

The average input results in much faster efficiency than a worst case input. Most numbers that are up to 30 digits long are composite with a relatively large number of small factors. The Pollard Rho algorithm runs in $O(\sqrt{p})$ where p is the smallest prime number factor of n. Therefore, Pollard Rho's algorithm will become

find a divisor very quickly ($\sim$ less than a second). Additionally, since the numbers will be composite, we will likely not need to run Pocklington's Algorithm. The efficiency will be around $O(|n|^2 \cdot \sqrt{p} \cdot k \cdot log^3(n))$, where k is the number of rounds of the Miller Rabin algorithm.

(d) Optimizations for algorithm performance

The naive solution to partition the string in all possible methods is exponential and strictly greater than $O(2^n)$. Since the program's input will be a number that contains up to 30 digits, the naive solution cannot be used to compute the result in a reasonable amount of time.

However, based on the exchange argument method, we proved that only key calculations using maximums are necessary to compute the correct answer. Therefore, I developed an iterative dp solution that utilized the previously computed maximum values.

Additionally, I created a faster solution by carefully selecting more efficient prime factorization and primality proving algorithms. The naive trial division approach to finding a prime factorization of n takes $O(\sqrt{n})$ time which is unreasonable for $n \leq 10^{31}$. Therefore, I employed a dual-pronged approach that utilizes trial division for removing smaller prime factors and Pollard Rho's algorithm for removing larger primes. I chose the Pollard Rho algorithm as it has an elegant implementation and is easily understandable.

I employed a creative technique to improve the efficiency of the deterministic primality proving algorithm. Firstly, I recognized that many "probabalistic" primality proving algorithms are actually "composite tests", which only err with psuedoprimes. A psuedoprime is a composite number that the algorithm treats as prime. Therefore, I could use these algorithms as a composite checker, and only deterministically prove primality of the numbers it returned as prime. The two primality proving tests I chose are the Baille-PSW test, a composite test which is deterministic for $n < 2^{64}$, and the Pocklington Test, a deterministic algorithm to run for $n > 2^{64}$.

In the future, I would use two different algorithms that employ elliptical curves: Lenstra's Elliptical Curve Method (LCM) and Elliptical Curve Primality Proving (ECPP), to further increase the efficiency of my solution. LCM is the fastest prime factorization method for 30 digit numbers. I would use the ECPP for primality proving since it works extremely fast for inputs that are up to thousands of digits long. I did not leverage these methods in my implementation since they both utilize higher level concepts which I have yet to become proficient in including Elliptical Curves and Abstract Algebra. Through learning these topics, I hope to greatly increase the efficiency of my algorithm in the future!

Problem 2

(a) Why your program always produces a correct solution

My solution uses dp to save all useful multiples at each index of the input string.

The solution has three important parts:

- **The stored state**
  All relevant multiples that can be formed at each index
- **The state transition**
  Create new multiples from the previously stored states. Multiply $dp[j]$ and substring from i to j.

$$dp[i] \text{ stores all } digit[j+1:i] \cdot dp[j]$$

- **The base case**
  The only value you can create is 1, with no digits.

$$dp[0] = 1$$

My algorithm always produces a correct answer as it employs a similar strategy as the naive solution. In contrast to the naive solution, which recursively creates a partition, my solution stores only relevant previous values.

(b) Worst Case Efficiency

The efficiency of my dp solution is $O(|n|^2 \cdot \sqrt{n} \cdot \sqrt[4]{n})$, where $|n|$ represents the length of the input string. The $O(|n|^2)$ term is due to an $O(|n|)$ nested loop within another $O(|n|)$ loop. I calculated the prime factorization with the Pollard Rho algorithm in $\sqrt[4]{n}$, for each substring formed by the indices within these two loops. I traversed through all of the multiples stored in previous states within these two loops. In order to avoid storing exponential number of possibilities, I only store factors of the final *product*. Therefore, I store a maximum of $O(\sqrt{n})$ factors of the number. The final Big O notation is $O(|n|^2 \cdot \sqrt{n} \cdot \sqrt[4]{n})$.

The algorithm is extremely efficient for numbers with fewer divisors and less efficient for numbers with a larger number of divisors.

(c) Average Case Efficiency

This algorithm is much more efficient for the average case than the worst case. The algorithm's rate limiting step is storing the numbers that are divisors of n, which in the worst case is $O(\sqrt{n})$. In the average case, however, n will have far fewer divisors and the efficiency will be greatly increased.

(d) Optimizations for algorithm performance

I used an efficient data structure and made several critical observations about the input to reduce the running time of the algorithm. The main improvements are listed below:

In a similar fashion to my solution to the last problem, I stored past possible multiples in previous states. Storing these values enabled me to efficiently build products from the past multiples.

Second, I realised that the stored multiplicands must be divisors of the *product*. Therefore, one only needs to save possibilities that are divisors of the *product*. This observation enables achieving polynomial instead of exponential run time.

Third, I keep all values in prime factored form instead of expanding the multiplication when determining the feasibility of creating the *product*. This form enables both the efficient recognition of the *product*'s divisors as well as faster multiplication with simple number theory tricks. Also, keeping numbers in prime factored form makes it much faster to compute when there is no possible solution, as the guidelines suggested.

Fourth, I used a hash map for efficient retrieval. We need to check if each value from previous states is a factor of the final *product*. This can be efficiently performed using a hash map in $O(1)$ time.