Computer Science Set

2020-11-30

Problem 1. Find the partition of a string of non-zero digits which produces the greatest combined number of prime divisors.

Why my program always gives a correct answer.

My program produces the correct answer because it considers all possible partitions and stores the combined prime divisors of viable partitions in an ArrayList before comparing the array size and printing out the partition with the greatest array size.

An optimization accurately decides which partition is a "viable" partition by estimating a "benchmark", a lower bound of the number of prime divisors, where if a partition's higher bound does not exceed this benchmark partition, then it is an nonviable partition. The "benchmark" estimation works like a prerequisite for a partition. If the partition is deemed viable, then it will continue on to have its prime divisors calculated for further inspection while a nonviable partition will exit early in the partition method.

By systematically checking each necessary partition, my program provides an accurate answer for each input.

What is the worst case efficiency of my program as $O(n)$ where n is the number of digits.

The first action my program does is factor the necessary partitioned numbers into their prime factors (More details in my factor memorization optimization description). Here, I factor half of a 2D array with length of n which totals to $\frac{n^2}{2}$ factoring.

The worst case scenario is if there is an input that is a product many small factors under 1000 and a large prime number. The number of small factors exceeds the lower bound of the "benchmark", thus forcing the program to factor the prime number completely (More details in my factor pruning optimization).

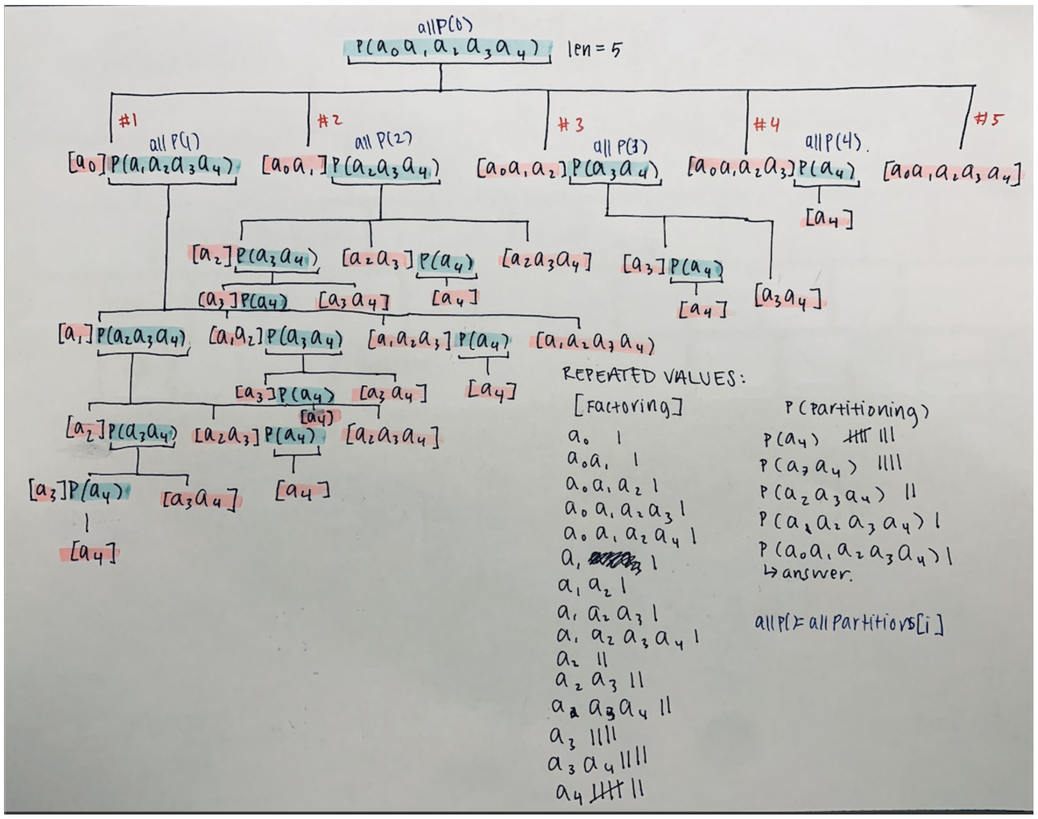
The number of times the program divides for a prime number is 1 to $\sqrt{number}$. The *number*, in terms of n, is 10^n. Thus, the efficiency of my program in terms of its worst case scenario is $O(10^{n/2}(\frac{n^2}{2}) \Rightarrow O(10^{n/2}(n^2))$.

What is the expected case when given a randomly chosen string of n.

My expected case is when the number *does* trigger all my optimizations. This means that, after prime factoring to 1000, none of the number's partitions are greater than my "benchmark" number. Here, each factorization to 1000 is a constant value, thus it does not need to be considered in terms of $O(n)$. However, my program's first step of factoring all the necessary partitioned numbers into their prime factors is still the same here as in the worst case scenario. As a result, our complexity is $\Rightarrow O(n^2)$.

Description and justification of optimizations

My initial non-optimized program was recursion. It used a divide and conquer algorithm which exhausted all possible partitions. The image below is a visual of this recursive method, where the input is a 5 digit number $a_0a_1a_2a_3a_4$



Notation

$a_1, a_2, ...a_n$	*Non-zero string of digits*
Brackets	*Definite partitions (to be factored immediately)*
$P(a_i...a_n)$	*Continue to Partition*
allP	*Array allPartitions used for a later optimization.*

This recursive method successfully returned the correct answer. However, it took hours to complete larger numbers. For example, a 25 digit number "573298273232432832938293" took around 16 hours. This is when I began my optimizations to increase the efficiency of my program.

1. Factor Memorization Optimization

   Justification:

   The problem with my initial recursion strategy was that there is a lot of redundancy ("Repeated values" I called them in my visual, with tally marks for each redundant call). In the example above, my program factored a_3a_4 four times.

   Description:

   To remove redundancy, we first calculated all the needed prime factorizations and stored it in a 2D array for later use (called allFactors). This way, the program does not have to factor large numbers multiple times if it has already done it.

   Later in my third optimization, I will create an algorithm which makes it so my program does not need to factorize all the numbers completely.

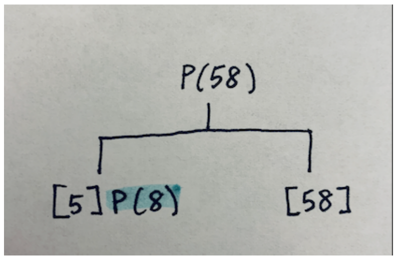
2. Bottom-Up Partition Optimization

   Justification:

   Same reasoning as my first optimization. My recursive program partitioned numbers more than once which significantly slowed down run time. For example, in the image my program partitioned $P(a_3a_4)$ four times.

   Description:

   Let's say the input is 58.



   Using the recursive tree, we see that in order to solve $P(58)$ you need to solve $P(8)$ first. With this in mind, I implemented a bottom-up algorithm with solved $P(a_n)$, then used $P(a_n)$'s result to solve $P(a_{n-1})$ and so on until $P(a_0a_1...a_n)$. After each partition, I stored the result in an array (called allPartitions), starting from index n to 0. Index 0 stores $P(a_0a_1...a_n)$, which is ultimately the answer to the problem.

3. Factor Pruning Optimization

   Justification:

   The program took around 2 hours to complete for 30 digit numbers. The majority of the time spent was in the prime factorization of large partitions. This optimization uses mathematical logic to prove that certain partitions will not have the greatest number of combined prime divisors before the program factors the number, thus allowing the program to bypass that factorization and move on.

   Description:

   Before the factor method calculates the primes factors, it will run an estimate method which returns a rough lower bound of how many divisors the number could possibly have if it were partitioned different and not kept unpartitioned. The estimate function works by iterating through the digits one by one, counting the number of prime divisors the single digit has. (Special case: If there is a 1 in the number, it is not partitioned to be alone since 1 has 0 prime divisors which is not ideal. Instead it is paired with the next digit). Each partitioned digit(s) is then factored, then summed together to create a "benchmark" partition.

   After finding the lower bound, the factor method will find and divide out all the prime factors up to 10^3 instead of to the square root of the number. Let's say for the number n digits, the number of factors it has under 10^3 is k. After it has been factored until 10^3, the left over number of digits the number has is m. At this point, any other prime divisor must be greater than 10^3, thus the maximum number of prime factors n has left is $\frac{m}{3}$.

If $\frac{m}{3} + k$ is not bigger than the benchmark partition, then the prime factorization of the whole number will not be considered only return the factors up to 10^3. If $\frac{m}{3} + k$ is bigger, then prime factorization will continue to 10^4, and $\frac{m}{4} + k$ will be considered and so on.

Problem 2. Find the partition of a string of non-zero digits which product is equals to a second number from 1 to the input.

Why my program always gives a correct answer.

My program always gives a correct answer because it exhausts every single partition possible and checks its product with the second input.

What is the worst case efficiency of my program as $O(n)$ where n is the number of digits.

The worst case efficiency of my program occurs for the first input $a_1a_2a_3...a_{n-1}a_n$ when the second input is divisible by $a_1, a_2, a_3...a_{n-1}$ but NOT a_n (basically all the first input's digits except the last one). This means that the program is forced to divide and check every partition branch until the very last possible partition, which will return "No Solution" since it is not a divisor. As a result, my program will divide the same number of times as there are number of partitions, excluding the last partition, which is $O(2^{n-1} - 1)$.

Note that this scenario is an extremely niche case, and likely an impossible situation. I suspect that there is a limit to the number of factors a number can have, thus preventing this program from running through the entire possible divisors.

What is the expected case when given a randomly chosen string of n

The expected case is that the program will only enter a few branches out of the n possible partition branches it checks. In terms of $O(n)$, the efficiency will be $O(n)$ with a constant.

Description and justification of optimizations

Similar to my first program, my program for this second problem used recursion. It branched off possible partitions in an identical format as the visual I provided in my explanation for my first program, except the bracketed values were used to divide against the second input rather than prime factored.

1. Shortening the Recursion Tail Optimization

   Justification:

   Although my recursion method proved to be fairly fast when I tested it, I knew that my program was not as efficient as it could be. For example, after a certain point in a partition branch, there is with certainty that it cannot be a solution. However, my initial program would continue on until it reaches the base case.

   Description:

   My program is similar to my first recursion branch image, where the brackets would be definite partitions and the $P(a_n)$ would be numbers that are still uncertain to be partitioned. My program decides whether or not it should continue down a partition branch if the definite partition value is a divisor of the second input. If it is not, then it is sure that this specific way is invalid.

   Another way this optimization cuts down the recursive tree is comparing the a_n value in $P(a_n)$ with the second input. If a_n is less than the second value, then the program will stop continuing down that branch. This is because it is impossible to get a partition of a number to have a combined product that is greater than the initial number.

   Proof:
   A number with n digits is partitioned in half (it can be partitioned any way, but for the sake of simplicity, it is partitioned only once).

The digits on the left comprise the number a and the digits on the right comprise the number b. There are d digits in b. Thus, the n digits can be written as $a * 10^d + b$.

The product of the partition is $a * b$. Since there are d digits in b, $b < 10^d$. If $b < 10^d$, then it is not possible to have $a * b >= a * 10^d + b$.

References

1 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Third Edition. The MIT Press, 3rd edition, 2009.

2 Schildt, Herbert. Java: The Complete Reference. 2019. Print.