Hamming Problem Report

Teja Dhondu

March 2023

1 Problem 1

1.1 Process

My first impression of the problem, based on the lack of a pattern in the sequence of numbers, was that it was some sort of counting problem. I felt that it was either some sort of greedy or dynamic programming problem. I spent some time trying to figure out if there was a substitution rule to creating a multiplier to generate new entries in the sequence from previous ones. After that I did some research online and learnt that the numbers are known as hamming numbers and that hamming numbers could be generated by multiplying previous numbers in the sequence with either 2, 3, or 5. The question was which numbers should be considered as candidates to multiply with 2, 3, or 5. Some algorithms online suggested considering every previous number as a candidate, placing them in a priority queue and selecting the least number at each step. This solution would work but it wasn't very space efficient.

Further research led me to a more efficient solution that suggested that 2, 3 and 5 would only have 1 candidate each at any point and that this candidate would obviously be the least possible number that they could be multiplied with to generate a new hamming number, since we want the least possible new number that can be generated. At each step, the candidate with the smallest value after being multiplied with the multiplier is chosen. As such, candidates to multiply with 2, 3 and 5 can be kept track of with a single pointer for each of 2, 3 and 5. These pointers would start at the first hamming number, 1, and move up the sequence one by one as candidates are chosen. One thing to keep track of would be duplicate candidates as there can be multiple paths to generating a new hamming number. For example, the hamming number 10 can be generated by multiplying 2 with the candidate 5 or 5 with the candidate 2. So during each iteration all the pointers need to be checked carefully and updated appropriately

1.2 Proof

The hamming number H can be represented as $2^i 3^j 5^k$. When $i > 0$ it can be generated by multiplying 2 with $2^{i-1}3^j 5^k$, which is a previously generated entry in the sequence. Similarly, when $j > 0$ it can be generated by multiplying 3 with $2^i 3^{j-1} 5^k$ and when $k > 0$ it can be generated by multiplying 5 with $2^i 3^j 5^{k-1}$. It is also obvious that 2, 3 and 5 would have only one candidate at any point in time and that candidate would be the least number possible because we want to generate the smallest possible new hamming number. If we did not generated the smallest possible hamming number then the sequence would not be in ascending order.

1.3 Performance

The time and space complexity are both O(N) since the algorithm loops N times, generating all N hamming numbers one by one in a constant time each loop and storing them. However, the limiting factor here is the representation of the hamming numbers. The hamming numbers very quickly blow up to large values which cannot be represented by mere 32 or 64 bit representations of decimal numbers. Trying to do so would lead to overflows. The decimal representation using the 64 bit data type can only store numbers up to around $N = O(10^4)$ without overflowing.

2 Problem 2

2.1 Process

The easiest part of the problem was figuring out that a representation using triplets could overcome the overflow issue of the decimal representation. The idea would be to represent the numbers as a triplet of values where the values are the exponents of 2, 3 and 5. This could comfortably represent the hamming numbers up to 10^{18} and possibly even beyond using the 64 bit data type. However, the issue with this representation was in how to compare the numbers without converting them to their decimal representations. This was necessary since we would have to calculate the minimum of the candidates at each step.

After thinking on the problem for a while an idea I came up with was to calculate the GCD of the numbers that needed to be compared and divide them by the GCD. This would reduce the value of the numbers, hopefully to a point where they can be represented by a decimal number, so that they could be compared. Additionally, the GCD would be very easy to calculate for the triplet representation. It is as simple as taking the minimum of each of the exponent values between the two numbers to be compared. Dividing the numbers by the GCD is as simple as subtracting the values of the exponents by the GCD.

However, I realized that this would not work in all cases as there would be many numbers for which the GCD would be 1 or a very small number. However, I left the GCD part in the code as it could still serve to speed up calculations in a few cases. I then performed some research online and came upon the idea of using logarithms to convert the hamming numbers to their log values, which would be small values even for large hamming numbers, and then comparing them. Additionally, computing the logs is extremely easy for the triplet representation. The rest of the algorithm is virtually identical to the first problem, with the only difference being the representation used for the numbers being the triplet representation instead of the decimal representation.

2.2 Proof

Given two hamming numbers $2^{i_1}3^{j_1}5^{k_1}$ and $2^{i_2}3^{j_2}5^{k_2}$ the GCD would be $2^i3^j5^k$ where i is the largest possible value such that 2^i divides 2^{i_1} and 2^{i_2}, j is the largest possible value such that 3^j divides 3^{j_1} and 3^{j_2}, and k is the largest possible value such that 5^k divides 5^{k_1} and 5^{k_2}. Clearly this means $i = min(i_1, i_2)$, $j = min(j_1, j_2)$, $k = min(k_1, k_2)$. Subtracting the exponents of both numbers by the GCD values will yield the result of dividing both numbers by the GCD.

The logarithm of a hamming number $2^i3^j5^k$ is calculated as $log(2^i3^j5^k) = log(2^i) + log(3^j) + log(5^k) = ilog(2) + jlog(3) + klog(5)$. It is evident that this equation is easy to use to calculate the log value of a hamming number in the triplet representation and therefore it is the equation used in the code.

2.3 Performance

The time and space complexity are O(N) just like in problem 1 but the limit of the values that the hamming numbers can take is greatly increased. The limit on the values that can be represented is now limited by the maximum number that can have its log represented with enough precision by the 64 bit floating point data type.

3 Problem 3

3.1 Process

Based on the fact that N was as large a number as $4 * 10^9$, it was clear that I would not be able to generate all N hamming numbers one by one as O(N) space and time complexity would take too much space and time respectively (approximately 100GB and 10 minutes). Some research online led me to some Haskell implementations, and an old blog post (`https://www.drdobbs.com/architecture-and-design/hamming-problem/228700538`) that explained them, where the idea was to generate only a slice of hamming numbers around the Nth hamming number. We use a formula to estimate the value of the Nth hamming number, and since it is an estimate we then use a corrective factor to generate a low and high bound within which the Nth hamming number would reside. We then enumerate all possible triplets that could represent hamming numbers within the low and high bounds. Each triplet would also give us information on how many hamming numbers are below the slice. After that, we sort the slice of triplets and taking the difference between N and the number of hamming numbers below the slice gives us the position of the Nth hamming number in the sorted slice.

3.2 Proof

The following formula from the Wikipedia entry for Regular Numbers gives us the number of Hamming numbers below a value n

$$N = \frac{(log(n\sqrt{30}))^3}{6log2log3log5}$$

Rearranging the formula gives us

$$log(n) = \sqrt[3]{6Nlog2log3log5} - log(\sqrt{30})$$

Which is the log estimate of the Nth hamming number. Since it is only an estimate, we can generate the high and low bounds by adding and subtracting the estimate with $\frac{1}{log(n)}$. It is important to note that we can only work with the log estimate instead of a direct estimate due to the limitations mentioned in the previous problem.

Let i, j, k be the triplet values representing the exponents of 2, 3 and 5 respectively, and hi and low be the bounds for the slice (Note: Only $log(hi)$ and $log(low)$ are available in the code). The possible values for k are from 0 to k_{hi}. $k_{hi} = \lfloor log_5(hi) \rfloor = \lfloor \frac{log(hi)}{log(5)} \rfloor$. For each k, the possible values of j are from 0 to j_{hi}. $j_{hi} = \lfloor log_3(hi - 5^k) \rfloor = \lfloor \frac{log(hi)-klog(5)}{log(3)} \rfloor$. Lastly, for each j and k, i can be between i_{low} and i_{hi}. Similarly to j, $i_{hi} = \lfloor \frac{log(hi)-klog(5)-jlog(3)}{log(2)} \rfloor$. Conversely, $i_{low} = \lceil \frac{log(low)-klog(5)-jlog(3)}{log(2)} \rceil$.

Triplets lesser than the lower bound are below the slice. For each combination of j and k, all triplets with an $i < i_{low}$ are below the slice. So by cumulatively counting all such i_{low} values for every j, k combination we obtain the number of hamming numbers below the slice. Sorting the slice of triplets is trivial since we have already solved how to compare the triplets.

3.3 Performance

The time and space complexity is based on the size of the slice. The size of the slice is O($N^{\frac{2}{3}}$). This is because we iterate over j and k (we don't count i here because it is a constant based on the corrective factor) which are proportional to

$log(n)$ which itself is proportional to $\sqrt[3]{N}$. This means we can run the algorithm for N equals to 1 trillion within 1 second.