Set Operations using Properties of Prime Numbers

Ori Yonay OYONAY12@TAMU.EDU
Department of Computer Science
Texas A&M University

Abstract

We present a simple algorithm for computing the union, intersection, and deletion of subsets of a larger set/multiset (for example, operations between subsets of words in the English language). We employ properties of products of prime numbers as a fast set/multiset fingerprinting mechanism. We show that while this algorithm achieves optimal asymptotic time complexity on certain problems and provides an interesting theoretical framework for computing a variety of operations on sets/multisets, it may not be practical for most applications. Our primary contribution is a theoretical framework for approaching the set/multiset data structure as well as a practical, efficient algorithm for certain tasks (mostly involving multisets of letters). We conclude by presenting algorithmic variants of the core idea on two competitive programming problems.

Keywords: Set (data structure), Prime Numbers

1. Introduction

Sets are some of the most heavily-used abstract data types in computer science due to their extremely fast ($O(1)$) running time on frequently-used operations with a hash table implementation. We present an interesting theoretical framework for approaching sets and set operations through simple mathematical operations involving prime numbers, as well as a practical algorithm for a small subset of tasks (mostly involving subsets/submultisets of letters in a word and small sets/multisets in general).

Note : throughout this paper we refer to operations on sets/subsets. The same algorithms and operations may similarly be applied to multisets/submultisets without loss of generality.

2. Algorithm

Let $S_1, S_2, ..., S_n$ be elements of our superset S containing n elements and P_k be the k^{th} prime number. We define the fingerprint f of a given subset Q of S with k elements as

$$f = \prod_{i=1}^{k} P_i$$

and 1 for the empty set. Since a prime factorization of the subset's fingerprint by definition yields all its prime factors (which represent set elements), the fingerprint may be thought of as a complete representation of the subset. In this context, we define the following set operations on a set fingerprint f:

Addition of element S_i to the subset:

$$f = f \cdot P_i$$

Removal of element S_i to the subset:

$$f = \frac{f}{P_i}$$

Determining whether the subset contains element S_i:

$$P_i \mid f$$

By the properties of prime number multiplication, we may extend this formulation for subset fingerprints f, g containing multiple elements:

Intersection of f and g:

$$\gcd(f, g)$$

Union of f and g:

$$\frac{f \cdot g}{\gcd(f, g)}$$

Removal of subset g from f:

$$f = \frac{f}{g}$$

Determining whether g is a subset of f:

$$g \mid f$$

Where $\gcd(f, g)$ is the greatest common divisor of fingerprints f, g (note that in the case of set union, division by the GCD is not necessary for correctness; it is done solely in order to keep fingerprints as manageable as possible). We may further define more complex set operations:

Set difference between f and g:

$$\frac{f}{\gcd(f, g)}$$

Symmetric difference between f and g:

$$\frac{f \cdot g}{(\gcd(f, g))^2}$$

Determining whether f and g are disjoint:

$$\gcd(f, g) = 1$$

Note that in the case of the symmetric difference $((f \cup g) - (f \cap g))$, division by the square of the GCD is in fact necessary for correctness, since it represents the removal of the intersection of f and g from their union.

We store a hash table indexed {element: i} and a list of the first n prime numbers (where n is the superset cardinality), which we use to perform the operations as described above. This allows us to extend our algorithm to support the introduction of new elements to our superset (i.e., introducing a new word to the English dictionary) by simply appending our list of primes with the next prime number ($O(\sqrt{n})$ by naive methods and sub-linear in general) and add the appropriate entry to our hash table.

3. Proof of Correctness

By the properties of prime number multiplication, any product of unique prime numbers will have no factors other than the prime numbers themselves. All of the operations above maintain the invariant that the set fingerprint f strictly contains (is the product of) the prime numbers corresponding to the appropriate set elements which should appear in f.

4. Efficiency and Practicality

Assuming integer multiplication and division is done in constant time, all set operations are $O(1)$ with the exception of operations involving GCD calculation. The running time of GCD is $O(\log f)$ for a set fingerprint f. Since P_n is $O(n \ln n) = O(n \log n)$ and a subset contains $O(n)$ elements, f is $O((n \ln n)^{O(n)})$. The time complexity of GCD on a fingerprint f is therefore $O(\log(O((n \log n)^{O(n)})) = O(n \log n)$. Therefore, all operations involving GCD calculation (intersection, union, symmetric difference) are $O(n \log n)$. These operations are $O(n)$ for hash tables.

Though in theory this provides a relatively easy-to-implement data structure for sets, it is in general impractical due to the relative inefficiency of numerical computations on very large numbers and the asymptotic growth of prime numbers. Suppose we wish to compute set operations on the entire English dictionary, which contains around $600,000$ words. The $600,000^{th}$ prime $8,960,453$, and the average P_i for this set is approximately 4.3 million. A subset containing even $50,000$ words would have a fingerprint on the order of $4,318,616^{50,000}$, which is a 331,768-digit number. This is far too large of a number to be efficiently computed, and a simple hash table will provide far better performance in practice.

Nevertheless, this algorithm does have practical uses in which it provides not only superior time complexity than any existing algorithm (to the best knowledge of the author) but also optimal complexity, which we discuss and prove in the following section.

5. Algorithmic Variations

In this section, we present two variations on our core idea to efficiently solve two distinct but similar competitive programming problems. We then prove that our algorithm is mathematically optimal for such problems.

5.1 Wordmaster

Wordmaster is a popular word game in which players are given six English letters and need to use them to create as many valid words as possible. For example, if the six letters were NLURAS then RUNS and LUNAR are valid words. Multiple uses of a letter are only allowed if there are multiple occurrences of that letter in the set of starting letters, i.e., ALL is an invalid word since L occurs only once, not twice, in our list of starting letters.

Suppose we are tasked with writing a program that outputs all valid words given a set of target letters and a list of every word in the English dictionary. More formally, for each word in the English dictionary, we would like to figure out whether every single letter of a given word is present in our list of seed letters, while accounting for duplicate letters.

Note that this is not a proper 'set' operation due to the fact that we care about letter occurrence frequency; the 'sets' may contain duplicate entries and are therefore not sets but rather multisets by definition. A useful property of our algorithm is that this is not a concern due to the fact that prime factorization of a product with multiple occurrences of the same prime number will contain exactly those occurrences; 7 and 49 are factors of 49, but $7^3 = 343$ is not.

One could easily solve this problem using the core idea presented here by assigning each letter a unique prime number, then computing the 'word score' as the product g of a given word's prime numbers associated with each letter. We calculate the same 'word score' f for the set of seed letters. The word is valid if and only if $g \mid f$. This calculation is $O(m)$ where m is the length of the given dictionary word. Applying this calculation for all words in the dictionary yields $O(nm)$ performance, where n is the number of words in the dictionary and m is the length of the longest word in the dictionary.

Algorithm 1 Wordmaster Algorithm

```
function WordScore(word)
    score ← 1
    for letter in word do
        i ← index of letter                    ▷ i.e., A yields 1, Z yields 26
        p ← i^th prime number
        score ← score · p
    end for
end function

function WordMaster(letters, words)
    f ← WordScore(letters)
    for word in words do
        score ← WordScore(word)
        if score | f then
            output word
        end if
    end for
end function
```

5.2 Largest Anagram Subset

We are given a list of words, and would like to find the largest subset of words which are anagrams of each other. For example, the set of words ['ant', 'magenta', 'magnate', 'tan', 'gnamate'] yields the subset ['magenta', 'magnate', 'gnamate'].

It directly follows from the communicative property of multiplication (and properties of prime numbers) that all (and only) anagrams of the same word will have the same 'word score', defined in the previous subsection as the product of each letter's associated unique prime number. Since 'word score' calculation is linear in the length of the word as shown above, we are able to calculate every word score in our list of words in $O(nm)$ time for n words and longest word length m. Finding the word score that occurs most frequently is

equivalent to mode calculation which is linear using a hash table, and outputting all words with the most frequent score is similarly linear. This yields $O(nm)$ running time.

Algorithm 2 Largest Anagram Subset Algorithm

```
function LargestAnagramSubset(words)
    scores ← []
    for word in words do
        score ← WordScore(word)
        append score to scores
    end for
    f ← most frequently occurring score in scores
    output every word whose associated score is f
end function
```

5.3 Proof of Optimality

We prove that our proposed algorithm is mathematically optimal for the two cases presented above. Since the core of both algorithms is the WordScore() routine to determine letter subsets (whether the letters of a given word is a subset of another multiset of letters), we prove its optimality first.

Lemma 1 *The lower bound on subset (and submultiset) detection is at least $O(m)$.*

Proof In order to correctly determine whether a set Q is a subset of some other set S, one must at least iterate through every single element in Q.

Proof by contradiction. Assume the above assumption is false, and there exists some algorithm which can perform subset detection without iterating through every element in Q. Then an adversary would be able to add some element not present (for multisets, not present as frequently) in S to Q. Under such an assumption this additional element is not guaranteed to be noticed by the algorithm, and therefore the algorithm cannot be correct. This contradicts our initial assumption of the existence of a sub-linear algorithm for subset detection. Therefore, subset detection is at least $O(m)$. ■

Since our algorithm only employs the WordScore() routine (which is $O(m)$ and a single modulus operation (which is $O(1)$) for subset detection, it is $O(m)$ and is therefore optimal. Mode computation has a lower bound of $O(n)$ and therefore Algorithm 2 achieves its lower bound of $O(nm)$. We must now only prove that the optimal Wordmaster algorithm has a lower bound of $O(nm)$.

Lemma 2 *The Wordmaster algorithm has a lower bound of $O(nm)$.*

Proof Any correct algorithm for both of the above examples requires iterating through every word in the given list (of which there are n), which takes $O(n) \cdot O(m) = O(nm)$ time.

Proof by contradiction. Assume some algorithm existed which did not iterate through every word in the given dictionary. An adversary would be able to append this list with

some word that is a subset of our list of target letters. Since not every word has been iterated upon, it is not guaranteed that the adversary's word (which is valid) will be considered and selected by this algorithm. Therefore this algorithm cannot be correct. This leads us to a contradiction of our initial assumption. Therefore, any proper Wordmaster algorithm is at least $O(nm)$. ■

6. Conclusion

We introduced an interesting algorithm for computing subset operations using elementary properties of prime numbers. We showed that while the algorithm is correct and mathematically more optimal than current algorithms on some problems, it is impractical due to the difficulty of efficient numerical computation on very large numbers and the inconveniently quick asymptotic growth of prime numbers. Two examples of modifications on the algorithm's core principles which are efficient in practice were provided, along with brief analysis of their running times and a proof of their optimality.