On fast transform algorithms for logical convolutions

TESINA

Author: Iván Alejandro Soto Velázquez

Supervisor: Benjamín Valdés Aguirre

ITESM CAMPUS QUERÉTARO

May 2018

Contents

Chapter 1

Introduction

1.1 Overview

The focus of this work will be on proving that a set of three transforms can efficiently compute XOR, OR, AND convolutions for any two vectors A and B. In mathematical notation, the operations called *logical convolutions* that will be examined can be expressed as follows:

XOR convolution

$$C_k = \sum_{i \oplus j = k} A_i B_j$$

OR convolution

$$C_k = \sum_{i \vee j = k} A_i B_j$$

AND convolution

$$C_k = \sum_{i \wedge j = k} A_i B_j$$

Moreover, three different selected problems that arise in a problem solving discipline called Competitive Programming [24] are discussed. These can be solved each with one of the logical convolutions. This approach along with alternative solutions and how they all compare in performance will be stated in future chapters.

p	q	p⊕q
0	0	0
0	1	1
1	0	1
1	1	0

p	q	p ∨ q
0	0	0
0	1	1
1	0	1
1	1	1

p	q	p ∧ q
0	0	0
0	1	0
1	0	0
1	1	1

Table 1.1: Truth tables for XOR, OR, AND operators from left to right

1.2 Preliminaries

Most of the definitions, theorems and proofs specific to the problem and solution at hand will be covered in later chapters, but it is advised to review the more general concepts which follow that are very common in Computer Science.

- Big $\mathcal{O}$ notation: In the analysis of algorithms, the big $\mathcal{O}$ notation is used to indicate the order or growth rate of a function. Mathematically, $f(x) = \mathcal{O}(g(x))$ if, for some constant C, and all sufficiently large values x that satisfy $x \geq x_0$ for a particular x_0, $f(x) \leq Cg(x)$ holds [13]. In less formal and more practical terms, the function $f(x)$ contained in $\mathcal{O}(f(x))$ indicates the worst-case number of steps an algorithm will take to finish, given that x is the input size to the procedure.

- Bit/binary digit: A bit is the unit of information used in computing. It can take only one of two values: 0 and 1.

- Logical operators: In Boolean algebra, each variable can take either two values: 0 or 1 (boolean variables are implemented as bits in computers). These variables can be evaluated under the binary operators $\oplus$ (XOR), $\vee$ (OR), $\wedge$ (AND).

  Each of the three operators act on two variables as indicated in the next tables, called *truth tables*. The variables are labeled p and q in the first and second column respectively, and the third column contains the result of the operator:

  Integers can be written as a sequence of bits, which is the same as expressing them in base-2. The logical operators can be applied to two integers resulting in a new integer. Each bit of the obtained number in a given position is the application of the operator to the bits in the first and second integer in the

same position. By reason of this behavior, the operators are called *bitwise operators*. A more formal on this operation definition is the following:

Definition 1.1. *Logical operation on integers: Let x and y be two non-negative integers. If x and y are less than $2^n - 1$, then they can be represented in binary notation with n coefficients that are either zero or one. Let*

$$x = \sum_{k=0}^{n-1} x_i 2^i$$

$$y = \sum_{k=0}^{n-1} y_i 2^i$$

be the extension of a number to its base 2 representation [23]. For each of the coefficients x_i and y_i their weight is provided by 2^i. Then any logical operation results in an integer z calculated as

$$z = \sum_{i=0}^{n-1} (x_i \ OP \ y_i) 2^i$$

where OP can be $\oplus$, $\vee$ or $\wedge$.

Chapter 2

State of the art

2.1 History of transforms and their computation

The study of transforms can be traced back to the history of linear algebra [20] and the birth of a field called *Transform theory* [21]. One of the transforms that is worth mentioning for the purpose of this work is the Discrete Fourier Transform (DFT). The DFT is widely used in Digital Signal Processing to obtain the spectrum of frequency content of a signal [19]. Historically, the importance of its application led to the creation of methods that are computationally efficient, which are often called "fast" algorithms [15]. Fast algorithms produce the same results as standard algorithms and perform better both in performance, accuracy and precision. The Fast Fourier Transform (FFT) is a very efficient algorithm for computing the DFT of a sequence of numbers. The published paper on it by Cooley and Tukey in 1965 [some reference] set a turning point in Digital Signal Processing [19]. It was shown that it was possible to calculate the DFT of a sequence of n numbers in time $\mathcal{O}(n \log n)$ over the standard algorithm that takes time $\mathcal{O}(n^2)$.

Later, efficient methods for all sorts of transforms were being published. For instance, the paper by Whelchel, Guinn in 1968 [27] introduces a fast algorithm for the computation of the Hadamard transform of a sequence of numbers. The Hadamard transform is a set of orthogonal functions called Walsh functions used to represent any discrete signal, just like a continuous signal can be represented by trigonometric functions [26]. The transform is also called the Walsh transform, Walsh-Hadamard transform, or Walsh-Fourier transform, all containing the same

set of functions in a different ordering. Currently, it has found applications in signal and image processing, speech processing, word recognition, signature verification, character recognition, pattern recognition, spectral analysis of linear systems, correlation and convolution, filtering, data compression, coding, communications, detection and statistical analysis [4].

The reasoning behind presenting the Walsh-Hadamard transform outstandingly is that it is the same transform for which XOR convolution is possible [10]. However, the focus of this work on the Walsh-Hadamard transform will be in a different light, but it is nevertheless appreciated to show a broader perspective on the mathematical tools covered.

The relevance of introducing the Discrete Fourier Transform earlier lies in its very close relationship to the properties of the Hadamard transform, but more importantly, on the similarities between the Fast Fourier Transform and the Fast Walsh-Hadamard transform for computation of the DFT and the Hadamard transform, respectively [27].

2.2 History of the convolution operation and the logical convolutions

The usefulness and applications of the convolution operation has been traced to areas such as applied mathematics, physics and engineering. As mentioned by Dominguez[14], almost none of the modern books discuss the theory and applications of the continuous convolution operation, despite its importance. Dominguez claims that the reason lies in that the discussion of the transforms themselves is the main topic of the books. However, the development and application of convolutions to various transforms can be roughly traced back in history. An important milestone took place in 1899, when French mathematician Borel obtained results related to the convolution theorem [14], which is key for this work and will be introduced later.

Even after such remarks, it is surprising to identify that there is very little discoverable work on the set of convolutions labeled *logical convolutions*. The use of the Fast Walsh-Hadamard transform for the computation of XOR convolution of two sequences [10] is the only well-documented source regarding this topic. A book

by Arndt [10] under section *"The Walsh transform and its relatives"* is one of the few sources that mention the OR convolution and the AND convolution. However, no interpretation of the transforms themselves and the correctness of the algorithms are provided; pseudocode for the fast algorithms is the only presentation.

Unpredictably, online sites are some of the readily available places where the XOR, OR and AND convolutions are mentioned. The online sites refer to programming challenges that can be solved using either of these operations in a desired time complexity. Sites such like the Chinese Software Developer Network (CSDN) present transforms to achieve different convolutions, but none of them reference any publication [6]. TopCoder, a community for designers, developers, data scientists, and algorithmists [1], provides a solution to one of its Single Round Match 518 [9] tasks that uses XOR convolution, but the claim in the solution sketch is that the transformation can be figured out by the contestants [9]. Additionally, other sites like Codeforces[7] and CSAcademy[5] have discussed the possibilities behind such transforms, but none of them reference any publication about them.

Chapter 3

Problem statement

As stated in the introduction, we would like to know how to calculate the following operations fast.

XOR convolution

$$C_k = \sum_{i \oplus j = k} A_i B_j$$

OR convolution

$$C_k = \sum_{i \vee j = k} A_i B_j$$

AND convolution

$$C_k = \sum_{i \wedge j = k} A_i B_j$$

for any vectors A and B resulting in vector C. For our purposes, a vector can be thought of as an array of numbers. For all scenarios in this work, we are restricting vectors to contain only **integer** numbers.

3.1 Definitions

It was said that a mathematical object called a *transform* made it possible to compute those fast, but its concept has not been introduced yet.

Definition 3.1. *A transform or transformation T over a domain D is a map that takes the elements $X \in D$ to elements $Y \in T(D)$, where the range of T is defined as $R(T) = T(D) = T(X) : X \in D$ [12]*

In other words, given two sets D and R composed of vectors, the transform T is responsible of assigning every vector in D to a vector in R. For instance, if we focus our attention on vectors of length n, both D and R contain vectors of length n, and we can make D contain all 2^n vectors and define T to be able to map them all. The transforms we will be covering can be represented as square matrices, that is, a table of values with number of rows equal to numbers of columns.

It is not yet clear how transforms can be useful for computing the desired operations. The next theorem will help out.

Theorem 3.2 (Convolution theorem)**.** *For a particular set of transforms, let T be one of them. It holds that*

$$T(A * B) = T(A) \odot T(B)$$

- The $*$ operator stands for the convolution of two vectors. A and B are said to be convolved.
- $\odot$ is the element-wise product of two vectors. The element-wise product of A and B, resulting in the vector $A \odot B$, is defined as

  $$(A \odot B)[k] = A_k B_k$$

  for every $k \in [0, n-1]$, where n is the length of vectors A and B (assuming they are of equal length, that is, $|A| = |B|$). In the definition, we care about the element-wise product of $T(A)$ and $T(B)$.

Finally, let's introduce one more definition.

Definition 3.3. *The inverse of a transform T is a transform T^{-1} such that $TT^{-1} = I$, where I is the identity matrix [12]. The identity matrix is a square matrix with only ones in its left-to-right diagonal and zeros everywhere else.*

If for a given T we also obtain T^{-1}, we can apply T^{-1} to both sides of the convolution theorem equation, thus cancelling T on the left side to obtain an expression for convolution in terms of the inverse transform of the element-wise multiplication of transforms of A and B.

The only missing thing is, are there transforms T such that, when applied to the convolution theorem, end up computing XOR, OR, AND convolution? Moreover, if they do exist, how can we transform vectors A and B with T on the right-side hand quickly? The next chapter will be covering the answers in detail to these questions. These answers are the main theoretical contribution of this work.

On the experimental side, it was mentioned in the introduction that we will be covering problems that can be solved with convolutions. If we can prove that convolutions can be computed fast, and we are able to model problem solutions as convolutions, we have gained a very powerful tool for problem solving.

Finding fast solutions is a common concern to hear about in problem solving disciplines, such as Competitive Programming. It turns out that for very particular problems, possibly one or more of its solution steps can be written as a mathematical expression. Computing the result of such formula as it is may be computationally slow, but if we get to identify it as a well-known expression, there is a chance that faster methods for its computation exist. This is exactly the scenario that happens with the problems that are going to be covered.

For each of the convolutions (XOR, OR, AND convolutions), a different problem will be described. Any context required to understand each problem will be given as well. The solutions to each problem will be delayed until the next chapters.

3.2 Problem for XOR convolution: Prime Nim

The problem that was picked to be solved with XOR convolution requires some prior notion of Combinatorial Game Theory. In case the reader is already familiar with it and the game of Nim, you can skip to the end of this subsection until the problem statement.

Combinatorial Game Theory (CGT) studies the strategies of two-player games of perfect knowledge and no chance moves, with a win-or-lose outcome[17]. For example: Chess, Go and Nim. We will focus on Nim, a combinatorial take-away game, since its mathematical model is simple yet powerful, and several two-player games can be reduced to Nim.

To be more precise, a combinatorial game, as defined by Ferguson[17], has to satisfy the following conditions:

1. There are two players.

2. There is a set, usually finite, of possible positions of the game.

3. The rules of the game specify for both players and each position which moves to other positions are legal moves. If the rules make no distinction between the players, that is if both players have the same options of moving from each position, the game is called impartial; otherwise, the game is called partizan.

4. The players alternate moving.

5. The game ends when a position is reached from which no moves are possible for the player whose turn it is to move. Under the normal play rule, the last player to move wins. Under the misere play rule, the last player to move loses.

6. The game ends in a finite number of moves no matter how it is played.

The game is said to be a take-away or substraction game, as every move reduces the game to a smaller instance, on which analysis can be done through backward induction (that is, from the end to the beginning of the game).

Nim: Two-player game where $k > 0$ piles are standing and each of those k has a positive number of pebbles. Each player in his turn has to pick one of the non-empty piles, and he has to take at least 1 pebble and at most the number of pebbles the pile has. Players move alternatively. The player who cannot move loses.

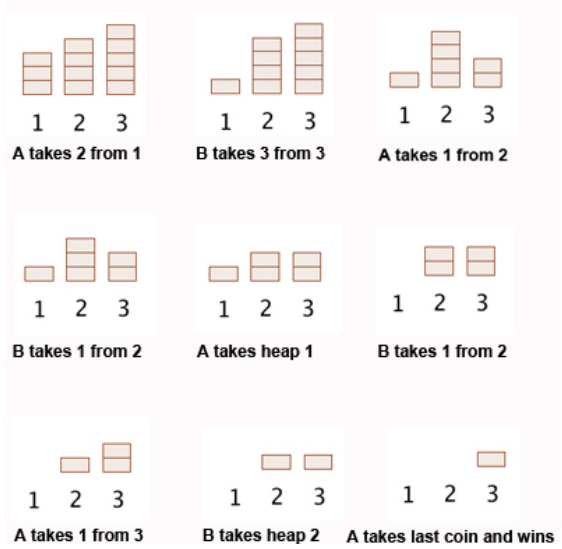
Positions can be divided into winning positions and losing positions:

1. The empty game, that is, the game where the current position has no moves, is a losing position.

2. A position is a winning position if there exists at least one move that leads the opponent to a losing position.

3. A position is a losing position if every move leads to the opponent's winning position.

Figure 3.1: A complete playthrough of Nim



It turns out that, given a Nim instance, there is a way to determine whether the player is in either a winning or a losing position. Before that, let's introduce a definition.

Definition 3.4. *The Nim-Sum of two non-negative integers is their addition without carry in base 2. That is, it is the equivalent to the* $\oplus$ *binary operation on two non-negative integers.*

Example. *Non-negative integers* 3_{10} *and* 7_{10} *in base 10 can be written in base 2 as* 0011_2 *and* 0111_2 *respectively. Then, the Nim-Sum of* 3_{10} *and* 7_{10} *is*

$$(0011_2) \oplus (0111_2) = 0100_2$$

Lemma 3.5. *A position is a winning position if the Nim-Sum of the piles is not zero. Otherwise, it is a losing position.*

Now that the ground has been set, this is the first of the problems we want to solve:

Problem. *Alice and Bob are about to play a game of Prime Nim. The only difference with normal Nim is that the size of each pile is a prime number. A prime number is a number that can only be divided by 1 and itself. There are*

$n > 2$ piles and each pile cannot be higher than h. Alice moves first. Bob is wondering about how many ways there are to set the n prime-length piles to win the game.
Constraints:

- $1 \leq n \leq 10^9$ – *the number of piles*
- $2 \leq h \leq 5 \times 10^4$ – *the upper bound on the height of any pile*

This problem was borrowed from the TopCoder Single Round Match 518. It was the hard problem of the Division 1 contest authored by Masaki Watanabe (user omeometo) [8].

For instance, let's say we would like to Bob how many ways Bob can win if we fix three piles. A few initial games would be:

3.3 Problem for OR convolution: MAXOR

Problem. *Given a set of numbers, we would like to find the maximum bitwise OR yielded by all pairs, and how many pairs generate this value.*
Constraints:

- $n \leq 10^5$ – *the size of the set*
- $S[i] < 2^{17}$ *for* $0 \leq i < n$ – *each element of the set*

This problem was taken from CSAcademy Round #53 [3].

Let's look at an example to grasp what the problem is asking:

Take for instance the set $\{3\ 7\ 4\ 11\ 2\ 13\}$. Some pair of values and their resulting OR value are next:

$$\{\mathbf{3}\ \mathbf{7}\ 4\ 11\ 2\ 13\} \rightarrow 3 \vee 7 = 7$$
$$\{3\ \mathbf{7}\ 4\ \mathbf{11}\ 2\ 13\} \rightarrow 7 \vee 11 = 15$$
$$\{3\ 7\ \mathbf{4}\ 11\ \mathbf{2}\ 13\} \rightarrow 4 \vee 2 = 6$$

If we try every pair of values, we will notice that the maximum OR that can be obtained is 15. In that case, we would like to know to know how for many pairs their OR value is 15. There are six pairs in total, which are the following:

$$\{\mathbf{3}\ 7\ 4\ 11\ 2\ \mathbf{13}\} \rightarrow 3 \vee 13 = 15$$
$$\{3\ \mathbf{7}\ 4\ \mathbf{11}\ 2\ 13\} \rightarrow 7 \vee 11 = 15$$
$$\{3\ \mathbf{7}\ 4\ 11\ 2\ \mathbf{13}\} \rightarrow 7 \vee 13 = 15$$
$$\{3\ 7\ \mathbf{4}\ \mathbf{11}\ 2\ 13\} \rightarrow 4 \vee 11 = 15$$
$$\{3\ 7\ 4\ \mathbf{11}\ 2\ \mathbf{13}\} \rightarrow 11 \vee 13 = 15$$
$$\{3\ 7\ 4\ 11\ \mathbf{2}\ \mathbf{13}\} \rightarrow 2 \vee 13 = 15$$

Therefore, the answer for this particular case is $(15, 6)$, corresponding to the maximum OR and the number of pairs with the maximum OR, respectively.

3.4 Problem for AND convolution: AND closure

Problem. *Given a set of numbers, we would like to find how many different values exist if we consider the bitwise AND of the numbers of every subset of this set. Constraints:*
$n \leq 10^5$ *– size of the set*
$S_i \leq 10^6$ *for every* $i \in [0, n-1]$ *–* S_i *is the ith element of the set S. This problem was taken from CSAcademy Round #13. It is authored by Alex Tatomir (user atatomir)[2].*

To clearly understand the problem, let's introduce what a subset is:

Definition 3.6. *A subset Y of a set X is a set that only contains some (possibly none or all) elements of X*

Now, let's take a look at some examples.

Let's say the given set is $\{3\ 7\ 4\ 11\ 2\ 13\}$. The size of this set is $n = 6$. In each of the next lines, a different subset is marked with a bolder color.

$$\{3\ \mathbf{7}\ 4\ \mathbf{11}\ 2\ 13\}$$
$$\{3\ \mathbf{7}\ 4\ 11\ \mathbf{2}\ \mathbf{13}\}$$
$$\{3\ 7\ \mathbf{4}\ 11\ 2\ 13\}$$
$$\{3\ \mathbf{7}\ \mathbf{4}\ 11\ 2\ 13\}$$

For the same subsets highlighted above, the bitwise AND of their values is shown:

$$\{3\ \mathbf{7}\ 4\ \mathbf{11}\ 2\ 13\} \rightarrow 7 \wedge 11 = 3$$
$$\{3\ \mathbf{7}\ 4\ 11\ \mathbf{2}\ \mathbf{13}\} \rightarrow 7 \wedge 2 \wedge 13 = 0$$
$$\{3\ 7\ \mathbf{4}\ 11\ 2\ 13\} \rightarrow 4 = 4$$
$$\{3\ \mathbf{7}\ \mathbf{4}\ 11\ 2\ 13\} \rightarrow 7 \wedge 4 = 4$$

Of course there are many more subsets (there are a total of 2^n of them), but hypothetically, if only these four were all the subsets, the answer to be problem would be 3, as that is the number of bitwise AND values. Considering all subsets, the answer for this particular case is 10.

Chapter 4

Proofs and solutions

We have not yet defined what is meant by efficient computation of the logical convolutions. It is easy to see (and will be briefly explained) how they can be computed in $\mathcal{O}(n^2)$. The claim is that, given that we can answer the questions set in the Problem Statement chapter, the resulting fast algorithms for computation in logical convolution will run in time $\mathcal{O}(n \log n)$.

Any evidence of a way to compute XOR convolution fast in any scope besides the competitive setting can actually be found in very few places [7] [9]. For the other two operations, proofs inspired by studying the proof of the Walsh-Hadamard transform for efficient XOR convolution will be presented. To the best of my knowledge, it is unlikely to find them anywhere, thus the core contribution from this paper lies in the composition of proofs of correctness for fast AND convolution, OR convolution, along with a detailed one for XOR convolution.

The next sections will cover proofs for XOR convolution, OR convolution and AND convolution. Additionally, we will go through the solutions of the problems stated in the previous chapter.

4.1 XOR convolution

Given two vectors $A = (a_0, a_1, ..., a_{n-1})$ and $B = (b_0, b_1, ..., b_{n-1})$ of length n, we would like to compute a new vector $C = (c_0, c_1, ..., c_{n-1})$ as follows:

$$C_k = \sum_{i \oplus j = k} A_i B_j$$

In other words, this convolution is just an operation to produce a new vector from two vectors (of the same length as both of them, assuming both are of equal length). The way this new vector is constructed is by fixing each of the positions of this vector (let's call the current position k ($k \in [0, n-1]$), and add up a number a from A, another number b from B, such that the XOR of the position of a in A and the position of b in B turns out to be k.

Of course, it is very easy to come up with a solution that iterates over every pair $(a_i, b_j) : a_i \in A$ and $b_j \in B$. For every such pair, we update the position $i \oplus j$ of C by adding $a_i b_j$ to it. The issue with this approach is that iterating over every such pair takes time $O(n^2)$. In case we are dealing with polynomials with relatively high degree bound ($n > 50000$, for example), this solution is too slow.

What we can do, and will do, is come up with a different way to calculate the same vector C. At first sight, the time complexity between the previous solution and the solution to be later shown seems to be the same, but we will discover that we can exploit some properties to achieve sub-quadratic time.

First of all, let's recall the **convolution theorem** presented some chapters ago.

Theorem 4.1 (Convolution theorem)**.** *There exists a subset of (linear) transforms (let's say T is any of the transforms in that subset), such that*

$$T(A * B) = T(A) \odot T(B)$$

holds true.

- The $*$ operator stands for the resulting convolution of two vectors. In this definition, A and B.
- $\odot$ is the element-wise product of two vectors. The element-wise product of A and B, resulting in the vector $A \odot B$, is defined as

$$(A \odot B)[k] = A_k B_k$$

for every $k \in [0, n-1]$, where n is the length of vectors A and B (assuming they are of equal length, that is, $|A| = |B|$). In the definition, we care about the element-wise product of $T(A)$ and $T(B)$.

For example, the Discrete Fourier Transform (DFT) is one of the transforms that satisfies theorem 4.1. But, the DFT calculates a convolution different from the XOR convolution presented in the beginning of the chapter. Namely, it computes a vector C as

$$C_k = \sum_{j}^{k} A_j B_{k-j}$$

where A and B are the vectors used in the convolution to compute C in every position k.

For the sake of clarity, computing that convolution using the DFT would be as follows:

- Choose vectors A and B. In case they are not of the same length, add padding zeros until they are of the same length.
- Let n be the new (larger) length of both vectors. Make n a power of 2. That is, n can be written as 2^k for some non-negative k. Add padding zeros to both A and B until their length is the same as n.
- The DFT, despite its fancy name, is a matrix. Just to follow the definition, the matrix T will be the name for the matrix DFT.
- To compute $T(A)$ and $T(B)$, we have to multiply $TA^\top$ and $TB^\top$ using the rules of matrix-vector multiplication. $A^\top$ and $B^\top$ are the transpose of vectors A and B respectively; that is, A and B are written upside-down.
- Once we have vectors $T(A)$ and $T(B)$, we can multiply them element by element and obtain a new vector $T(A) \odot T(B)$.
- Because of the convolution theorem, we equivalently obtained $T(A * B)$. We care about the convolution, not the transformed convolution.

- Given another matrix T^{-1} such that $TT^{-1} = I$, where I is the identity matrix (1 in the main diagonal, 0 everywhere else), we can multiply T^{-1} and $T(A*B)$ to obtain $A * B$. That is, $T^{-1}T(A * B) = A * B$.

It is well known that matrix-vector multiplication takes time $\mathcal{O}(n^2)$, which we used to explain how transforms are applied to vectors, and time complexity-wise, that is not any better than the aforementioned solution for XOR convolution. Nevertheless, the DFT has a property that allows for execution of the previous steps in time $\mathcal{O}(n \log n)$.

For XOR convolution, the DFT is not of much interest, but the important thing to notice is that, if we can find a matrix that also satisfies theorem 4.1, with a nice property that allows for a faster computation, we can compute convolution just like we did with the DFT!

(Note: It won't be explained how the DFT does it in $\mathcal{O}(n \log n)$. Instead, we will see how the to-be-presented transform for XOR convolution can achieve the same time complexity)

4.1.1 Proof for a XOR convolution transform

To devise a transform that can do XOR convolution, we will start with small matrices that correctly compute XOR convolution, and then prove that this result can be generalized.

For instance, let's say that the vectors A and B, from which we would like to compute $A * B$, are both of length $n = 1$ ($A = (a_0)$ and $B = (b_0)$). As an important remainder, the length of the resulting convolution vector, $C = (c_0)$, will be the same as that of A, which should be the same as that of vector B. Computing for $n = 1$ is trivial:

$$c_0 = a_0 b_0$$

.

This is clearly true, because $0 \oplus 0 = 0$, and vector C is updated at position 0.

Since the point of introducing the convolution theorem is using it, what sort of transform T can produce XOR convolution, for this particular case?

It can be seen that this case does not even require a transform at all. If we ignore the use of a transform that multiplies the vectors:

$$A * B = A \odot B$$

holds true for $n = 1$. Therefore, our transform can simply output its vector input when the length of the input vector is $n = 1$. That is,

$T(X) = X$ if $|X| = 1$ for some vector X.

If that is the case, then

$$T(A * B) = T(A) \odot T(B)$$

is true as well. Since we care about $A * B$, we need to undo the T applied to $A * B$. To achieve that, we can let $T^{-1}(X) = X$, where T^{-1} is the inverse matrix of T. In other words, T and T^{-1} are identity transforms, since $T(X) = X = T^{-1}(X)$.

What about the case where both vectors have length $n = 2$? In effect, $A = (a_0, a_1)$ and $B = (b_0, b_1)$. If we inspect this scenario, how can A and B be used to calculate C?

a_0 and b_0 are both at position 0. The multiplication of both of them will be added to c_0. The same goes for a_1 and b_1, since $1 \oplus 1 = 0$. To add to c_1, we add $a_0 b_1$ and $a_1 b_0$ since the XOR of their positions is 1. Put in simple notation:

$$c_0 = a_0 b_0 + a_1 b_1$$

$$c_1 = a_0 b_1 + a_1 b_0$$

If we had the appropriate transform for XOR convolution, inputting the vector (c_0, c_1) from the c values just calculated, it should be the same as multiplying $T((a_0, a_1))$ and $T((b_0, b_1))$.

It turns out the following transform works for the $n = 2$ case.

$$T((a_0, a_1)) = (a_0 + a_1, a_0 - a_1) \tag{4.1}$$

or written as a $2x2$ matrix: $\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$

Proof.

$$\begin{aligned}
T(A * B) &= T((c_0, c_1)) \\
&= T((a_0b_0 + a_1b_1, a_0b_1 + a_1b_0)) && \text{(express both } c \text{ in terms of } a \text{ and } b) \\
&= ((a_0b_0 + a_1b_1 + a_0b_1 + a_1b_0, a_0b_0 + a_1b_1 - a_0b_1 - a_1b_0) \\
& && \text{(apply transform 4.1)} \\
&= ((a_0 + a_1) \cdot (b_0 + b_1), (a_0 - a_1) \cdot (b_0 - b_1)) && \text{(group terms)} \\
&= (a_0 + a_1, a_0 - a_1) \odot (b_0 + b_1, b_0 - b_1) \\
& && \text{(separate to a element-wise vector multiplication)} \\
&= T((a_0, a_1)) \odot T((b_0, b_1)) && \text{(undo transform 4.1)} \\
&= T(A) \odot T(B) && \text{(write } (a_0, a_1) \text{ as A and } (b_0, b_1) \text{ as B)}
\end{aligned}$$

□

It was mentioned, with no emphasis at all, that the transforms for which the convolution theorem hold, are linear. A transform T is linear if it satisfies the following

$$T(A + B) = T(A) + T(B)$$

for any vectors A and B

$$T(cA) = cT(A)$$

for any vector A and constant c

For the purpose of the upcoming proofs, we only care about proving the first identity.

Lemma 4.2 (Linearity of T)**.** *The* $2x2$ *transform* T *satisfies* $T(A + B) = T(A) + T(B)$

Proof.

$$
\begin{aligned}
T(A+B) &= T((a_0,a_1)+(b_0,b_1)) \\
&\qquad\qquad \text{(rewrite A and B as } (a_0,a_1) \text{ and } (b_0,b_1)) \\
&= T((a_0+b_0,a_1+b_1)) \qquad \text{(add both vectors)} \\
&= (a_0+b_0+a_1+b_1, a_0+b_0-(a_1+b_1)) \qquad \text{(use 4.1)} \\
&= (a_0+a_1, a_0-a_1)+(b_0+b_1, b_0-b_1) \\
&\qquad\qquad \text{(separate into two vectors)} \\
&= T(A)+T(B) \qquad \text{(undo 4.1)}
\end{aligned}
$$

□

So far, we have two key results: the convolution theorem holds for this transform when the length of both vectors is $n = 2$. We also proved that $T(A + B) = T(A) + T(B)$, which will turn out to be useful next.

To prove for larger vectors, we will focus only on lengths that are powers of two. A vector can always have length $n = 2^k$ for some non-negative integer k by adding zeros at the end, which clearly do not affect the result.

The transform we used for the $n = 2$ is designed to work for vectors of that length. If we can build a transform that depends on this one, we can proceed to use the results we have just proved. Naturally, a transform that can be built recursively can be further investigated to see if it meets the desired behavior. Since it is assumed that $n = 2^k$, it is obvious that n will be divisible by 2. Because every vector A can be split in two halves (A_1 and A_2 for the first and second half, respectively) and written as another vector with both ordered parts ($A = (A_1, A_2)$), the following transform can handle them:

$$T((A_1, A_2)) = (T(A_1) + T(A_2), T(A_1) - T(A_2)) \tag{4.2}$$

In other words, a vector A transformed by T can be recursively built by transforming its halves, A_1, A_2, add them for the first entry, and subtract them for the second entry.

Let's show it holds for $n = 2$

Proof.

$$\begin{aligned} A &= (a_0, a_1) \\ T((A_1, A_2)) &= T((a_0, a_1)) \\ &= (T(a_0) + T(a_1), T(a_0) - T(a_1)) \end{aligned}$$

Since we know that for the trivial case for $n = 1$, $T(x) = T^{-1}(x) = x$ for any element x

$$= (a_0 + a_1, a_0 - a_1)$$

which is the same convolution theorem as that resulting from the previous transform. Therefore, we can use this transform without invalidating the proofs for $n = 2$. We can now forget about the special case, and let T take this definition from now on.

□

As we are working with a vector length $n > 2$, the linearity of T and its validity for the Convolution theorem must be shown. Fortunately, we can use a "Proof by induction", to show that if, assuming it holds true for a smaller value than n, it will hold true for n, and therefore it will be true for any n, as we can trace the proof back to the small case of $n = 2$.

Lemma 4.3 (Linearity of T)**.** *For the transform T, $T(A + B) = T(A) + T(B)$*

Proof. A can be written as (A_1, A_2), the same for B as (B_1, B_2)

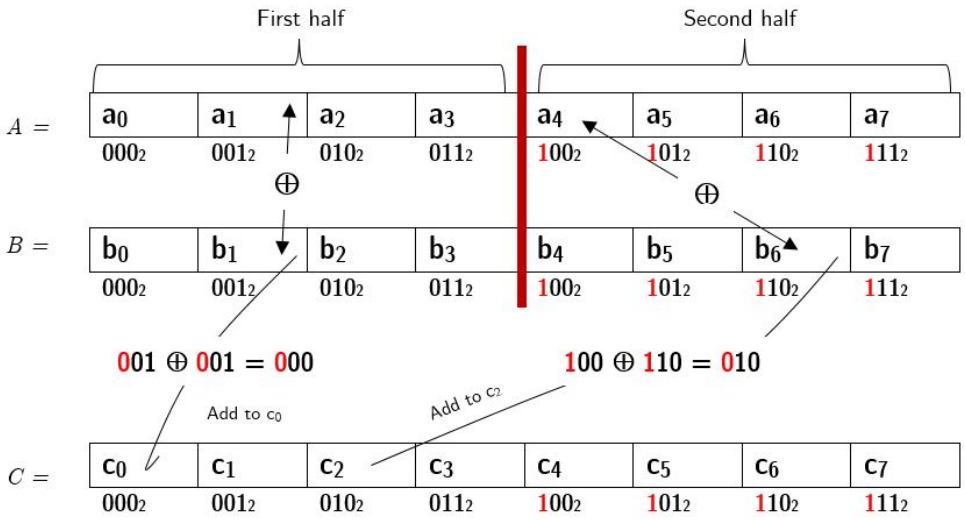
$$
\begin{aligned}
T(A+B) &= T((A_1, A_2) + (B_1, B_2)) && \text{(split } A \text{ and } B \text{ in two halves each)} \\
&= T((A_1 + B_1), (A_2 + B_2)) && \text{(add the vectors)} \\
&= (T(A_1 + B_1) + T(A_2 + B_2), T(A_1 + B_1) - T(A_2 + B_2)) && \text{(use Equation 4.1)} \\
&= (T(A_1) + T(B_1) + T(A_2) + T(B_2), T(A_1) + T(B_1) - (T(A_2) + T(B_2))) \\
& && \text{(knowing that Lemma 4.9 holds)} \\
&= (T(A_1) + T(A_2), T(A_1) - T(A_2)) + (T(B_1) + T(B_2), T(B_1) - T(B_2)) \\
& && \text{(separate as addition of vectors)} \\
&= T((A_1, A_2)) + T((B_1, B_2)) && \text{(undo Equation 4.1)} \\
&= T(A) + T(B) && \text{(join both halves into } A \text{ and } B \text{ respectively)}
\end{aligned}
$$

□

The importance of proving linearity is seen as we prove the **XOR convolution theorem** for T. But, before that, a key observation is necessary to proceed.

Observation 4.4 (Behavior of XOR operation)**.** *Given that n has to be a power of two, and $A = (a_0, a_1, ..., a_{n-1})$, $B = (b_0, b_1, ..., b_{n-1})$, $A_1 = (a_0, a_1, ..., a_{\frac{n}{2}-1})$, $A_2 = (a_{\frac{n}{2}}, a_{\frac{n}{2}+1}, ..., a_{n-1})$, $B_1 = (b_0, b_1, ..., b_{\frac{n}{2}-1})$, $B_2 = (b_{\frac{n}{2}}, b_{\frac{n}{2}+1}, ..., b_{n-1})$, let's think about the following: if XOR is applied to any pair of numbers belonging to the first $\frac{n}{2}$ positions of A and any of the first $\frac{n}{2}$ positions of B, the resulting number will be less than n, because positions are as big as $\frac{n}{2}$, and since n is a power of two, $\frac{n}{2}$ is also a power of 2. That means that the highest position in the first half, $\frac{n}{2} - 1$, has all k bits on in its binary representation, where $k = log_2(\frac{n}{2})$. Thus, the XOR of any two numbers between 0 and $\frac{n}{2} - 1$ cannot be any bigger than $\frac{n}{2} - 1$, as both numbers have at most the first k bits on. The same thing happens with positions from $\frac{n}{2}$ to $n-1$. It is the case of $[0, \frac{n}{2} - 1]$ but with the $k+1_{th}$ bit on. The XOR of any two numbers in that range will turn off that bit, so the result will lie between 0 and $\frac{n}{2} - 1$. For the rest of the cases, we can match a position in $[0, \frac{n}{2} - 1]$ coming from A with a position in $[\frac{n}{2}, n - 1]$ coming from B, or vice versa. One has the $k + 1_{th}$ bit on, while the other does not, so the number resulting from the XOR*

Figure 4.1: Visualization of the behavior of the XOR operation between same vector halves. Indices of vectors are labelled using binary numbers to illustrate Observation 4.4.



of both positions will be in the interval $[\frac{n}{2}, n-1]$. *As it was an even split of the positions, positions in* $[0, \frac{n}{2}-1]$ *lie in* A_1 *and* B_1, *and positions in* $[\frac{n}{2}, n-1]$ *in* A_2 *and* B_2. *Both* $A_1 * B_1$ *and* $A_2 * B_2$ *added up set the first half of the convolution, and* $A_1 * B_2$ *added up with* $A_2 * B_1$ *the second half. Figure 4.1.1 provides a graphic visualization of this behavior.*

Theorem 4.5 (XOR convolution theorem for T)**.** $T(A * B) = T(A) \odot T(B)$

Proof.

$$
\begin{aligned}
T(A * B) &= T((A_1, A_2) * (B_1, B_2)) && \text{(split } A \text{ and } B \text{ in two halves)} \\
&= T((A_1 * B_1) + (A_2 * B_2), (A_1 * B_2) + (A_2 * B_1)) && \text{(apply Observation 4.4)} \\
&= (T(A_1 * B_1 + A_2 * B_2) + T(A_1 * B_2 + A_2 * B_1), \\
&\quad T(A_1 * B_1 + A_2 * B_2) - T(A_1 * B_2 + A_2 * B_1)) && \text{(use Equation 4.2)} \\
&= (T(A_1 * B_1) + T(A_2 * B_2) + T(A_1 * B_2) + T(A_2 * B_1), T(A_1 * B_1) + \\
&\quad T(A_2 * B_2) - T(A_1 * B_2) - T(A_2 * B_1)) && \text{(use Lemma 4.3)} \\
&= (T(A_1) \odot T(B_1) + T(A_2) \odot T(B_2) + T(A_1) \odot T(B_2) + T(A_2) \odot T(B_1), \\
&\quad T(A_1) \odot T(B_1) + T(A_2) \odot T(B_2) - T(A_1) \odot T(B_2) - T(A_2) \odot T(B_1)) \\
& && \text{(inductive step on } A_1 * B_1) \\
&= (T(A_1)[T(B_1) + T(B_2)] + T(A_2)[T(B_1) + T(B_2)], \\
&\quad T(A_1)[T(B_1) - T(B_2)] - T(A_2)[T(B_1) - T(B_2)]) && \text{(group terms once)} \\
&= ([T(A_1) + T(A_2)][T(B_1) + T(B_2)], [T(A_1) - T(A_2)][T(B_1) - T(B_2)]) \\
& && \text{(group once again)} \\
&= (T(A_1) + T(A_2), T(A_1) - T(A_2)) \odot (T(B_1) + T(B_2), T(B_1) - T(B_2)) \\
& && \text{(separate as element-wise product of vectors)} \\
&= T((A_1, A_2)) \odot T((B_1, B_2)) && \text{(undo Equation 4.2)} \\
&= T(A) \odot T(B) && \text{(join the two halves as } A \text{ and } B)
\end{aligned}
$$

□

This is all that is needed to guarantee that XOR convolution is computed using T.

If the transform T is once again viewed as a matrix, it can be noticed that, to create T_n for length n, $T_{\frac{n}{2}}$ is required. More precisely, T_n is composed of four copies for $T_{\frac{n}{2}}$ like this:

$$T_n = \begin{bmatrix} T_{\frac{n}{2}} & T_{\frac{n}{2}} \\ T_{\frac{n}{2}} & -T_{\frac{n}{2}} \end{bmatrix} \tag{4.3}$$

It turns out this matrix is the Hadamard matrix, which can be constructed using the aforementioned recursive definition.

The importance of identifying this transform as the Hadamard matrix lies in the existing methods to calculate the Walsh-Hadamard transform (they are the same). As the Discrete Fourier Transform can be computed in time $\mathcal{O}(n \log n)$ using the Fast Fourier Transform (FFT), a variation of the FFT, the Fast Walsh-Hadamard Transform (FWHT), computes the Walsh-Hadamard transform of a vector in time $\mathcal{O}(n \log n)$. Moreover, the inverse of the Hadamard matrix is required to obtain the convolution.

The inverse of a Hadamard matrix H is given by:

$$H_n{}^{-1} = \frac{1}{n} H_n$$

and there is no need to prove all of the previous results for this matrix, as it can be shown that it is indeed the inverse of H_n.

Finally, we can show a procedure for fast computation of the Walsh-Hadamard transform. The following pseudocode is a recursive version of the FWHT

Algorithm 1 Recursive Fast Walsh-Hadamard Transform

procedure FWHT(A, invert)

$n \leftarrow |A|$

if $n == 1$ **then**

return

$A_1 \leftarrow A[0, \frac{n}{2} - 1]$

$A_2 \leftarrow A[\frac{n}{2}, n - 1]$

FWHT(A_1, invert)

FWHT(A_2, invert)

for $i = 0$ upto $\frac{n}{2}$ **do**

$A_i = A_{1_i} + A_{2_i}$

$A_{i+\frac{n}{2}} = A_{1_i} - A_{2_i}$

if invert **then**

$$A_i = \frac{A_i}{2}$$

$$A_{i+\frac{n}{2}} = \frac{A_{i+\frac{n}{2}}}{2}$$

In the FWHT recursive procedure, we can see that vector A is split into two vectors A_1 and A_2 of half the length of $|A|$, and then FWHT is called for both of them. Since we are assuming that the length of A is a power of two, A can be split into two evenly at every step of our procedure. The only occasion we cannot divide the length evenly into two is when we have are left with a single element, but at the beginning of this section, it was proved that the transform of a single element is itself, thus we can successfully stop recursion.

After calling FWHT for both A_1 and A_2, both vectors have to be somehow added to build A. We showed in Equation 4.2 how it is done. In $\frac{n}{2}$ steps, A is built from A_1 and A_2, and we may divide every element by two in case we are applying the inverse transform. It was claimed that the inverse transform really divides every element by n (here, n is the initial full length of A), but the

recursion-depth will perform division by two the required numbers of times to end up dividing by n at the end of the entire procedure.

More precisely, how many recursive calls are being executed? At every call, two additional calls are made, both of which with vectors of half the length. Since it was said that n is a power of two, and that we stop when there is only a single element, because A gets smaller by a factor of two, the number of calls to FWHT is $\mathcal{O}(n \log n)$.

Finally, since $\mathcal{O}(n)$ steps are performed in each call to update A (n here is the current length of A) for a total of $\mathcal{O}(\log n)$ levels, we can see that the time complexity of the algorithm is $\mathcal{O}(n \log n)$. As a fresh reminder, the slow solution's time complexity is $\mathcal{O}(n^2)$, so this different approach clearly reduces the number of operations performed.

Now that we have a procedure for the Walsh-Hadamard transform, the next procedure computes the convolution of two vectors

Algorithm 2 XOR convolution of two vectors

procedure FAST-XOR-CONVOLUTION(A, B)

$n \leftarrow 1$

while $n < max(|A|, |B|)$ **do**

$n \leftarrow n * 2$

$C \leftarrow vector(n)$

while $|A| < n$ **do**

$A \leftarrow (A, 0)$

while $|B| < n$ **do**

$B \leftarrow (B, 0)$

FWHT(A, *false*) {*false* because we apply the original transform to A}

FWHT(B, *false*) {same with B}

for $i = 0$ upto n **do**

$C_i = A_i B_i$

FWHT(C, *true*) {*true* because we apply the inverse transform to C}

return C

4.1.2 Solution for Prime Nim

4.1.2.1 Problem statement

Recalling the problem introduced in Chapter 3:

Problem. *Alice and Bob are about to play a game of Prime Nim. The only difference with normal Nim is that the size of each pile is a prime number. A prime number is a number that can only be divided by 1 and itself. There are $n > 2$ piles and each pile cannot be higher than h. Alice moves first. Bob is wondering about how many ways there are to set the n prime-length piles to win*

the game.
Constraints:

- $1 \leq n \leq 10^9$ – *the number of piles*
- $2 \leq h \leq 5 \times 10^4$ – *the upper bound on the height of any pile*

4.1.2.2 Solution

First of all, how would an easy-to-come-up solution would look like? If we know that the highest prime number we can consider is h, and the game's initial configuration has n piles, we could try every possible h^n game, and check if Bob wins. Of course, that is too slow!

To solve this problem fast, the key observations can be divided as:

- A nim-sum of 0 means we are in a losing position. Since we care about the number of ways Bob wins, it is equivalent to count the number of ways Alice loses. Alice loses if her nim-sum is 0, so it suffices to count how many ways we can create prime piles whose XOR is 0.
- What if there are only $n = 2$ prime piles? The problem is then asking to count the number of pile pairs such that their XOR is 0

We can figure that, if we build a characteristic vector A for the prime piles (that is, a vector A with 1 at every position k such that a pile of height h exist, and 0 otherwise), then running the following nested loops solves the problem (slowly, in time $\mathcal{O}(h^2)$):

$\{A$ is the characteristic vector$\}$

$answer \leftarrow 0$

for $i = 0$ upto h **do**

  for $j = 0$ upto h **do**

    if $i \oplus j == 0$ **then**

      $answer \leftarrow answer + A_i A_j$

Since we want to count every combination of k prime piles, the characteristic vector A should contain 1 in every index that is a prime number (up to the largest height a pile can have).

XOR convolution can be interpreted in this scenario as the number of pairs whose XOR of positions lie in a non-negative integer $k \in [0, n-1]$ (if there are n piles, indexed starting from 0). Since we showed that the Fast Walsh-Hadamard computes XOR convolution in time $\mathcal{O}(n \log n)$, it can efficiently solve the $n = 2$ case.

What to do when $n > 2$? Thinking about the result of XOR convolution carefully, we can see that if the XOR convolution of $A * A$ (let's store it in B) is once again convolved with A, that is, $B * A$, we have now calculated how many triplets take on each of the XOR values in $[0, n-1]$. For our final answer, we only care about the convolution at index 0, but the rest are needed to keep running this algorithm to produce a correct answer.

If we do the same again, we will know how many tuples of size 4, size 5, size 6, and so on, have a XOR value for each possible index. But, there is a small issue. What if the number of prime numbers is relatively big? Let's say, $n = 100000$, and the greatest pile can take the value $h = 5 \times 10^4$? If we do XOR convolution in such fashion (a total of n steps), we end up with an algorithm that runs in time $\mathcal{O}(nh \log h)$.

The observation we can make is that, we want to compute the answer for the given n, and we may or may not need every value $n' < n$.

One way is that we could obtain the answer for n by obtaining the answer for only $\log n$ smaller values. If we express n as a sum of powers of two:

$$n = (2^k \wedge n) + (2^{k-1} \wedge n) + ... + 2^0 \wedge n)$$

We can compute the answer for the $\mathcal{O}(\log n)$ powers of two that $\leq n$, and update our answer (by adding more "number of ways") only when a given power of two is part of the binary representation of n. This technique is called **binary exponentiation**.

In other words, we end up calling FAST-XOR-CONVOLUTION like this (for $h = 2$, for example):

FAST-XOR-CONVOLUTION(FAST-XOR-CONVOLUTION(A, B))

For any h value, this can be rewritten as:

$$\text{FAST-XOR-CONVOLUTION(A, A)}^h$$

But, instead of doing the call h times, $\log h$ calls can be done like this:

$$\text{FAST-XOR-CONVOLUTION(A, A)} * \text{FAST-XOR-CONVOLUTION(A, A)}^2 * \text{FAST-XOR-CONVOLUTION(A, A)}^4 *...* \text{FAST-XOR-CONVOLUTION(A, A)}^{2^k}$$

Then, the efficient algorithm, that runs in time $\mathcal{O}(h \log h \log n)$ will look like this:

Algorithm 3 Efficient solution for Prime Nim

procedure PRIME-NIM(A)

{A is the characteristic vector}

{B is the resulting convolution from every power of two less than n}

{C is the convolution that will contain the final answer for n}

$B \leftarrow A$

$C \leftarrow vector(|A|)$

while $n > 0$ **do**

if $n \wedge 1$ **then**

$C \leftarrow$FAST-XOR-CONVOLUTION(C, B)

$B \leftarrow$FAST-XOR-CONVOLUTION(B, B)

$n \leftarrow \frac{n}{2}$

return C_0

4.2 OR convolution

The case with OR convolution (and even later, with AND convolution) is very similar to XOR convolution. As we have set ground for what has to be done, we can skip a lot of background information. For the remaining proofs, comments for steps will be omitted, as they are very highly similar to those found in the procedure for proving the transform for XOR convolution.

The OR convolution for two vectors A and B is defined as

$$C_k = \sum_{i \vee j = k} A_i B_j$$

A different transform from the Walsh-Hadamard transform is required to perform OR convolution. Nevertheless, an almost identical transform is used! This section will be used to prove linearity and the validity for the Convolution Theorem for this new transform.

4.2.1 Proof for a OR convolution transform

The 2×2 transform for OR convolution is

$$T = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \tag{4.4}$$

The only different between the H_2 Hadamard matrix and this matrix lies in the bottom-right number, now a 0 instead of -1.

As it was the case for $n = 1$ in the XOR convolution, there is no need for a transform to compute OR convolution. So $T(x) = T^{-1}(x) = x$. Sanity check before we proceed: let $C = (c_0)$ be the convolution of $A = (a_0)$ and $B = (b_0)$

$$c_0 = a_0 b_0$$

Clearly, $0 \vee 0 = 0$. Using no transform, we get the correct result, so it is safe to set the transform of A and B to output themselves.

Now, for the case of $n = 2$, it is identical to the scenario with XOR convolution.

The only thing that changes is the transform:

$$T((a_0, a_1)) = (a_0 + a_1, a_0)$$

(It is the same as the matrix representation shown earlier).

Lemma 4.6. $T(A + B) = T(A) + T(B)$

Proof.

$$\begin{aligned} T(A + B) &= T((a_0, a_1) + (b_0, b_1)) \\ &= (a_0 + a_1, a_0) + (b_0 + b_1, b_0) \\ &= T((a_0, a_1)) + T((b_0, b_1)) \\ &= T(A) + T(B) \end{aligned}$$

□

If we inspect how $A = (a_0, a_1)$ and $B = (b_0, b_1)$ calculate $C = (c_0, c_1)$, we end up with the following:

$$c_0 = a_0 b_0$$

$$c_1 = a_0 b_1 + a_1 b_0 + a_1 b_1$$

Theorem 4.7 (Convolution theorem for T)**.** $T(A * B) = T(A) \odot T(B)$

Proof.

$$\begin{aligned} T(A * B) &= T((c_0, c_1)) \\ &= (c_0 + c_1, c_0) \\ &= (a_0 b_0 + a_0 b_1 + a_1 b_0 + a_1 b_1, a_0 b_0) \\ &= ([a_0 + a_1][b_0 + b_1], a_0 b_0) \\ &= (a_0 + a_1, a_0) \odot (b_0 + b_1, b_0) \\ &= T(A) \odot T(B) \end{aligned}$$

□

As before, we now take a look at the case when $n > 2$ and n is a power of two. The transform will now be

$$T((A_1, A_2)) = (T(A_1) + T(A_2), T(A_1))$$

Lemma 4.8 (Linearity of T)**.** $T(A + B) = T(A) = T(B)$

Proof.

$$\begin{aligned}
T(A + B) &= T((A_1, A_2) + (B_1, B_2)) \\
&= T((A_1 + B_1, A_2 + B_2)) \\
&= (T(A_1 + B_1) + T(A_2 + B_2), T(A_1 + B_1)) \\
&= (T(A_1) + T(B_1) + T(A_2) + T(B_2), T(A_1) + T(B_1)) \\
&= (T(A_1) + T(A_2), T(A_1)) + (T(B_1) + T(B_2), T(B_1)) \\
&= T((A_1, A_2)) + T((B_1, B_2)) \\
&= T(A) + T(B)
\end{aligned}$$

□

Now, we can proceed to show that $T(A * B) = T(A) \odot T(B)$

Proof.

$$\begin{aligned}
T(A * B) &= T((A_1, A_2) * (B_1, B_2)) \\
&= T((A_1 * B_1), (A_1 * B_2) + (A_2 * B_1) + (A_2 * B_2)) \\
&= (T(A_1 * B_1) + T(A_1 * B_2) + T(A_2 * B_1) + T(A_2 * B_2), T(A_1 * B_1)) \\
&= (T(A_1) \odot T(B_1) + T(A_1) \odot T(B_2) + T(A_2) \odot T(B_1) + T(A_2) \odot T(B_2), \\
&\quad T(A_1) \odot T(B_1)) \\
&= ([T(A_1) + T(A_2)][T(B_1) + T(B_2)], [T(A_1)][T(B_1)]) \\
&= (T(A_1) + T(A_2), T(A_1)) \odot (T(B_1) + T(B_2), T(B_1)) \\
&= T((A_1, A_2)) \odot T((B_1, B_2)) \\
&= T(A) \odot T(B)
\end{aligned}$$

∎

□

Just like the Walsh-Hadamard transform, T_n is built from $T_{\frac{n}{2}}$. In matrix form, this is expressed as

$$T_n = \begin{bmatrix} T_{\frac{n}{2}} & T_{\frac{n}{2}} \\ T_{\frac{n}{2}} & 0_{\frac{n}{2}} \end{bmatrix} \tag{4.5}$$

Because of the very close similarity, the Fast Walsh-Hadamard can be slightly tweaked for computation using this transform in $\mathcal{O}(n \log n)$.

The inverse for T is

$$T_n{}^{-1} = \begin{bmatrix} T_{\frac{n}{2}}{}^{-1} & -T_{\frac{n}{2}}{}^{-1} \\ T_{\frac{n}{2}}{}^{-1} & 0_{\frac{n}{2}} \end{bmatrix} \tag{4.6}$$

and for the base case

$$T_2{}^{-1} = \begin{bmatrix} 1 & -1 \\ 1 & 0 \end{bmatrix} \tag{4.7}$$

Algorithm 4 Recursive Fast OR Transform

procedure FORT(A, invert)

$n \leftarrow |A|$

if $n == 1$ **then**

return

$A_1 \leftarrow A[0, \frac{n}{2} - 1]$

$A_2 \leftarrow A[\frac{n}{2}, n - 1]$

FORT(A_1, invert)

FORT(A_2, invert)

for $i = 0$ upto $\frac{n}{2}$ **do**

$A_i = A_{1_i} + (invert? -1 : 1)A_{2_i}$

$A_{i+\frac{n}{2}} = A_{1_i}$

The algorithm for OR convolution is really the same as that for XOR convolution, except we change the calls to the *fort* function instead.

Algorithm 5 OR convolution of two vectors

procedure FAST-OR-CONVOLUTION(A, B)

$n \leftarrow 1$

while $n < max(|A|, |B|)$ **do**

$n \leftarrow 2n$

$C \leftarrow vector(n)$

while $|A| < n$ **do** $A \leftarrow (A, 0)$

while $|B| < n$ **do** $B \leftarrow (B, 0)$

FORT(A, *false*)

FORT(B, *false*)

for $i = 0$ upto n **do**

$C_i = A_i B_i$

FORT(C, *true*)

return C

4.2.2 Solution for MAXOR

4.2.2.1 Problem statement

The next statement belongs to the problem introduced in the previous chapter:

Problem. *Given a set of numbers, we would like to find the maximum bitwise OR yielded by all pairs, and how many pairs generate this value.*
Constraints:

- $n \leq 10^5$ – *the size of the set*
- $S[i] < 2^{17}$ *for* $0 \leq i < n$ – *each element of the set*

4.2.2.2 Solution

The most straightforward approach to solve this is brute force: try every pair, compute their OR, and keep variables on the maximum OR found and how many pairs result in such OR. This would seem a reasonable solution if not for the n value that can be up to 10^5.

As it was hinted at the beginning, this problem has a solution using OR convolution. As a fresh reminder, this is the OR convolution operation (using programming indexing):

$$C[k] = \sum_{i \vee j = k} A[i]B[j]$$

If we take the solution hint, there should be some way to use the above expression to solve our problem, or at least significant part of it. We notice that the OR operation is happening to the indices of vectors A and B, and we need to take the OR operation of elements in S, not their indices in S. Moreover, we can see that the values $A[i]$ and $B[j]$ are multiplied and their result is added to $C[i \vee j]$, but once again, we only care about applying OR to values, not multiplying them. We could then, use a different representation for S, such that every $s \in S$ is an index instead of a value, and the value at index s has something useful in it.

Let's try this new representation: since every $S[i] < 2^{17}$, creating a new vector A of length 2^{17} is reasonable. Vector A should be another representation of S, which means that S could be restored from A. For every $s \in S$, set $A[s] = 1$, and for every $s \notin S$, set $A[s] = 0$. A represents S in what is called the *characteristic vector of S*.

Let's now use OR convolution with vector A. The result yields vector C:

$$C[k] = \sum_{i \vee j = k} A[i]A[j]$$

Values of S are now expressed in the indices, and there is a 1 at every index if the value is present in S. If both $A[i]$ and $A[j]$ are 1, that is equivalent to both i and j occurring in S. Multiplying 1 times 1 yields 1, and adding it to $C[i \vee j]$ is indeed counting a pair (i, j) whose OR is $i \vee j$. At the end, $C[k]$ will contain the number of pairs whose OR value is k, which is very useful for our purposes.

To find which is the maximum OR, we need to find the last k for which $C[k] \neq 0$. We said $C[k]$ contains how many pairs generate this value, but there is one small issue: it is also counting a number at some position of S being paired with itself at the same position, and such pairs should not be counted. Additionally, each pair is counted twice: pairs (i, j) and (j, i) are each counted, for some $i \in S, j \in S$; we only need one of them in our answer.

Nevertheless, there is enough information to correctly eliminate these occurrences. Clearly, when we take the OR of a number with itself, we obtain the same number. Then, given the highest k for which $C[k] \neq 0$, we need to count how many s values exist such that $s \in S$ and $s = k$. We proceed to remove these occurrences from $C[k]$. Finally, each pair is counted twice, so it suffices to divide $C[k]$ by two, and we are done. In short,

$$C[k] = \frac{(C[k] - occurrence[k])}{2}$$

where $occurrence[k]$ is the number of $k \in S$. Our solution should output pair $(C[k], k)$, indicating the number of pairs whose OR value is k, and k – the maximum OR yielded by all pairs, respectively.

4.3 AND convolution

The last of the operations that will be covered is AND convolution. We will go to through the same procedure as with XOR convolution and OR convolution. As a fresh reminder, the next mathematical expression corresponds to AND convolution.

$$C_k = \sum_{i \wedge j = k} A_i B_j$$

4.3.1 Proof for a AND convolution transform

Once again, for the $n = 1$ case, $T(a_0) = T^{-1}(a_0) = a_0$, since the AND operation at both positions 0 is 0.

For $n = 2$, the following matrix works:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \tag{4.8}$$

This matrix can also be expressed as $T((a_0, a_1) = (a_1, a_0 + a_1)$ Once again, let's investigate how vectors $A = (a_0, a_1)$ and $B = (b_0, b_1)$ cooperate to calculate $C = (c_0, c_1)$.

$$c_0 = a_0b_0 + a_0b_1 + a_1b_0$$

$$c_1 = a_1b_1$$

Before anything else, let's prove linearity

Lemma 4.9 (Linearity of T)**.** $T(A + B) = T(A) + T(B)$

Proof.

$$\begin{aligned} T(A + B) &= T((a_0, a_1) + (b_0, b_1)) \\ &= T((a_0 + b_0, a_1 + b_1)) \end{aligned}$$

Apply the transform 4.9

$$\begin{aligned} &= (a_1 + b_1, a_0 + b_0 + a_1 + b_1) \\ &= (a_1, a_0 + a_1) + (b_1, b_0 + b_1) \\ &= T((a_0, a_1)) + T((b_0, b_1)) \\ &= T(A) + T(B) \end{aligned}$$

□

Theorem 4.10 (Convolution theorem for T)**.** $T(A * B) = T(A) \odot T(B)$

Proof.

$$\begin{aligned} T(A * B) &= T((c_0, c_1)) \\ &= T((a_0b_0 + a_0b_1 + a_1b_0, a_1b_1)) \end{aligned}$$

Apply the transform

$$\begin{aligned}
&= (a_1 b_1, a_0 b_0 + a_0 b_1 + a_1 b_0 + a_1 b_1) \\
&= (a_1 b_1, [a_0 + a_1][b_0 + b_1]) \\
&= (a_1, a_0 + a_1) \odot (b_1, b_0 + b_1) \\
&= T((a_0, a_1)) \odot (T(b_0, b_1)) \\
&= T(A) \odot T(B)
\end{aligned}$$

□

For any other power for two bigger than 2, the transform is now

$$T((A_1, A_2)) = (T(A_2), T(A_1) + T(A_2))$$

Lemma 4.11 (Linearity of T)**.** $T(A + B) = T(A) + T(B)$

Proof.

$$\begin{aligned}
T(A + B) &= T((A_1, A_2) + (B_1, B_2)) \\
&= T((A_1 + B_1, A_2 + B_2)) \\
&= (T(A_2 + B_2), T(A_1 + B_1) + T(A_2 + B_2))
\end{aligned}$$

Induction step on A_i and B_i for $i \in \{1, 2\}$ with respect to linearity

$$\begin{aligned}
&= (T(A_2) + T(B_2), T(A_1) + T(B_1) + T(A_2) + T(B_2)) \\
&= (T(A_2), T(A_1) + T(A_2)) + T(B_2), T(B_1) + T(B_2)) \\
&= (T(A_2), T(A_1 + A_2)) + (T(B_2), T(B_1 + B_2)) \\
&= T((A_1, A_2)) + T((B_1, B_2)) \\
&= T(A) + T(B)
\end{aligned}$$

□

Theorem 4.12 (Convolution theorem for T)**.** $T(A * B) = T(A) \odot T(B)$

Proof.

$$T(A * B) = T((A_1, A_2) * (B_1, B_2))$$

Notice that only the two second halves (A_2 and B_2) will contribute to the convolution from $\frac{n}{2}$ to $n-1$. The rest will add from position 0 to $\frac{n}{2}-1$. Then

$$= T(A_1 * B_1 + A_1 * B_2 + A_2 * B_1, A_2 * B_2)$$
$$= (T(A_2 * B_2), T(A_1 * B_1 + A_1 * B_2 + A_2 * B_1) + T(A_2 * B_2))$$

Induction step for linearity

$$= (T(A_2 * B_2), T(A_1 * B_1) + T(A_1 * B_2) + T(A_2 * B_1) + T(A_2 * B_2))$$

Induction step on validity of the convolution theorem

$$= (T(A_2) \odot T(B_2), T(A_1) \odot T(B_1)$$
$$+ T(A_1) \odot T(B_2) + T(A_2) \odot T(B_1) + T(A_2) \odot T(B_2))$$
$$= (T(A_2) \odot T(B_2), [T(A_1) + T(A_2)][T(B_1) + T(B_2)]$$
$$= T(A_2), T(A_1) + T(A_2)) \odot (T(B_2), T(B_1) + T(B_2))$$
$$= T((A_1, A_2)) \odot T((B_1, B_2))$$
$$= T(A) \odot T(B)$$

□

This transform, in matrix form, can be defined recursively as follows:

$$T_n = \begin{bmatrix} 0_{\frac{n}{2}} & T_{\frac{n}{2}} \\ T_{\frac{n}{2}} & T_{\frac{n}{2}} \end{bmatrix} \tag{4.9}$$

Unlike the Walsh-Hadamard transform, the inverse of T_n is different from T_n multiplied by a real number.The inverse of T_n is defined as

$${T_n}^{-1} = \begin{bmatrix} -T_{\frac{n}{2}} & T_{\frac{n}{2}} \\ T_{\frac{n}{2}} & 0_{\frac{n}{2}} \end{bmatrix} \tag{4.10}$$

Finally, the procedures to compute this transform follow

Algorithm 6 Recursive Fast AND Transform

procedure FANDT(A, invert)

$n \leftarrow |A|$

if $n == 1$ **then**

return

$A_1 \leftarrow A[0, \frac{n}{2} - 1]$

$A_2 \leftarrow A[\frac{n}{2}, n - 1]$

FANDT(A_1, invert)

FANDT(A_2, invert)

for $i = 0$ upto $\frac{n}{2}$ **do**

if *invert* **then**

$A_i = A_{2_i} - A_{1_i}$

$A_{i+\frac{n}{2}} = A_{1_i}$

else

$A_i = A_{2_i}$

$A_{i+\frac{n}{2}} = A_{1_i} + A_{2_i}$

The algorithm for OR convolution is really the same as that for XOR convolution, except we change the calls to the BFWHT function instead.

Algorithm 7 AND convolution of two vectors

procedure FAST-AND-CONVOLUTION(A, B)

$n \leftarrow 1$

while $n < max(|A|, |B|)$ **do**

$n \leftarrow 2n$

$C \leftarrow vector(n)$

while $|A| < n$ **do** $A \leftarrow (A, 0)$

while $|B| < n$ **do** $B \leftarrow (B, 0)$

FANDT(A, *false*)

FANDT(B, *false*)

for $i = 0$ upto n **do**

$C_i = A_i B_i$

FANDT(C, *true*)

return C

4.3.2 Solution for AND closure

4.3.2.1 Problem statement

As a fresh reminder, this is the statement of the problem we want to solve:

Problem. *Given a set of numbers, we would like to find how many different values exist if we consider the bitwise AND of the numbers of every subset of this set.*
Constraints:
$n \leq 10^5$ *– size of the set*
$S_i \leq 10^6$ *for every* $i \in [0, n-1]$ *–* S_i *is the ith element of the set S.*

4.3.2.2 Solution

The easiest approach to this problem would use brute force: try every subset and calculate their AND value. Instead, we will come up with a few observations to understand how AND convolution can be applied to this problem.

For a moment, let's go back to find out how AND convolution looked like (using programming indexing):

$$C[k] = \sum_{i \wedge j = k} A[i]B[j]$$

We notice that the AND bitwise operation in AND convolution is happening in the indices of vector A and B, while multiplication is happening to the values in those positions. If i and j belonged to set S, $C[i \wedge j]$ could say something about them, and we care about such value because $i \wedge j$ is precisely the bitwise AND result of a size-two subset of S. Since we somehow need to take advantage of the AND operation happening in the indices, it would be handy to convert S to a new representation that solved our problem.

First, let's notice that every element in the set S is at most 10^6. Creating a vector of size 10^6 and iterating over it would be computationally reasonable – both in memory and time respectively. Let's call such vector A. For every element $s \in S$, set $A[s] = 1$ and $A[s] = 0$ otherwise. Vector A can also be referred to as the *characteristic vector of S*.

What would happen if A is convoluted with itself? That is, a vector C is obtained from:

$$C[k] = \sum_{i \wedge j = k} A[i]A[j]$$

Since the multiplication of $A[i]$ and $A[j]$ is stored at $C[i \wedge j]$ and A has only 1 and 0, an increment to $C[i \wedge j]$ would mean that both i and j are present in S, because they were set to 1 in A. After a single AND convolution operation, $C[k]$ would store for how many pairs of numbers in S the value k is their resulting bitwise AND, for every $k \in [0, n-1]$. It should be noted that, the pairs considered will include a number and itself in the same position for every number in S, because the bitwise AND with itself is itself, and there is no restriction in the convolution operation for considering a number and itself. Then, so far C has all size-1 and

size-2 subsets in S stored. We only care about the existence of a certain AND value, so it is just sufficient to check if the kth entry of C is different than zero to increment our answer.

The problem, though, asks for all subsets, including the empty subset. What we should do is next is to apply the same idea used to solve the Nim problem: Let's say C is the result of $A * B$. Let's obtain a new vector D by convolving C and A, i.e. $D = C * A$. D would contain, for every $k \in [0, n-1]$, the number of size-1, size-2 and size-3 subsets such that the AND of their elements is k. If we then convolve D and A to obtain E, we would include size-4 subsets as well. Finally, we can do this n times to find out for how many subsets, up to size-n subsets, their AND value is k. n is sufficient since the biggest subset of S has size n, which is itself. Since convolution takes $\mathcal{O}(n \log n)$ and it is done n times, the time complexity of this solution is $\mathcal{O}(n^2 \log n)$. Unfortunately, n can be up to 10^5, so this would be too slow.

Just like the solution to the Nim problem, we can apply binary exponentiation to perform only $\mathcal{O}(\log n)$ operations. The resulting time complexity would be $\mathcal{O}(n \log^2 n)$, which is fast enough to run under 3 seconds, assuming a single thread can perform approximately 10^8 operations in a second.

Chapter 5

Performance results and alternative solutions

To show that the Fast XOR, OR and AND convolutions were faster in speed than the brute force solutions in practice, experiments that consisted of running solutions to the previous problems for the purpose of comparing their runtimes took place. The rest of this chapter will cover, for each of the three problems, different correct solutions that perform differently. There are surprising results, as very particular solutions that are also fast exist besides those covered.

5.1 Results for problem Prime Nim

For this problem, there was only one optimal solution in terms of speed, and it was the solution explained a few chapters back. As a reminder, the techniques that allowed for a complexity of $\mathcal{O}(h \log h \log n)$ were:

- Fast Walsh-Hadamard Transform
- Binary exponentiation

To show how crucial these two optimizations are, solution that lack either or both of those were implemented and ran against various inputs. In other words, a solution that only used the Fast Walsh-Hadamard transform but added every

new pile one by one was implemented, as well as a Dynamic Programming solution that computes the answer for every game.

Every input the solutions ran against consisted of a pair of values $nandh$ - the number of desired piles and the maximum height of a pile. For the purpose of understanding the behavior of the function growth in time with respect to input size, the multiple input were fed in non-decreasing order for both n and h, in powers of two. For example, the solutions first ran against $n = 2, h = 2$, later $n = 4, h = 4$ and so on, until the input size $n = 10^9, h = 10^5$. Some solutions were not able to handle large inputs, because of memory and time restrictions, so time and memory upper limits were set for them. In case those were exceeded, the program was set to run for about 10 seconds, as to indicate it would run forever. Since for these solutions, time and memory complexity were correlated, it was reasonable to assume infinite running time whether any of the time or memory restrictions were violated.

A sketch of the solutions and their time and memory complexity is as follows:

- Fast-Walsh Hadamard transform and binary exponentiation: This was already described, but briefly it was about building a characteristic vector for the possible games consisting of a single pile, and proceeding to obtain the possible games for n piles using binary exponentiation.
  The time complexity of this solution is $\mathcal{O}(h \log h \log n)$

- Fast-Walsh Hadamard transform and naive multiplication: The only difference from this solution to the previous one is that, instead of computing $\log n$ convolutions using binary exponentiation, n convolutions are computed. Such different actually has a huge impact in the running time as n gets bigger.
  The time complexity of this solution is $\mathcal{O}(nh \log h)$

- Dynamic Programming: This is sort of a brute force, because it lacks both the fast transform and the binary exponentiation optimizations, but it is much better than creating h^n games and checking for each of them if Bob wins. The dynamic programming solution calculates the number of ways the nim-sum of the piles is j if we have considered i piles. The following recursion

performs such computation:

$$dp[i][j] = \begin{cases} 1 & i = 0 \& is\_prime(j) = 1 \& \ j \leq h \\ 0 & i = 0 \& is\_prime(j) = 0 \\ \sum_{p \oplus k} dp[i-1][k] & is\_prime(p) = 1 \& \ p \leq h \& \ k \leq \{\min_{2^b} : h \leq 2^b\} \end{cases}$$

At the end, the answer can be found at $dp[n][0]$, since the nim-sum has to be 0 for Bob to win, and that is after we have considered n piles in all games. This pseudo-polynomial time algorithm runs in $\mathcal{O}(nh^2)$ and requires $\mathcal{O}(nh)$ memory. Clearly, as soon as the values in the input get larger, problems in both time and memory occur.

It has not been mentioned yet, but it is possible to implement every fast transform in an iterative fashion. The recursive solutions were presented as they were more natural, but it turns out that a bottom-up approach is also possible for their computation. This is important as recursion can be much slower than iteration, even if both perform the same number of steps. The following graph shows the difference iteration over recursion can have over this problem, and additionally the rest of the suboptimal solutions are shown:

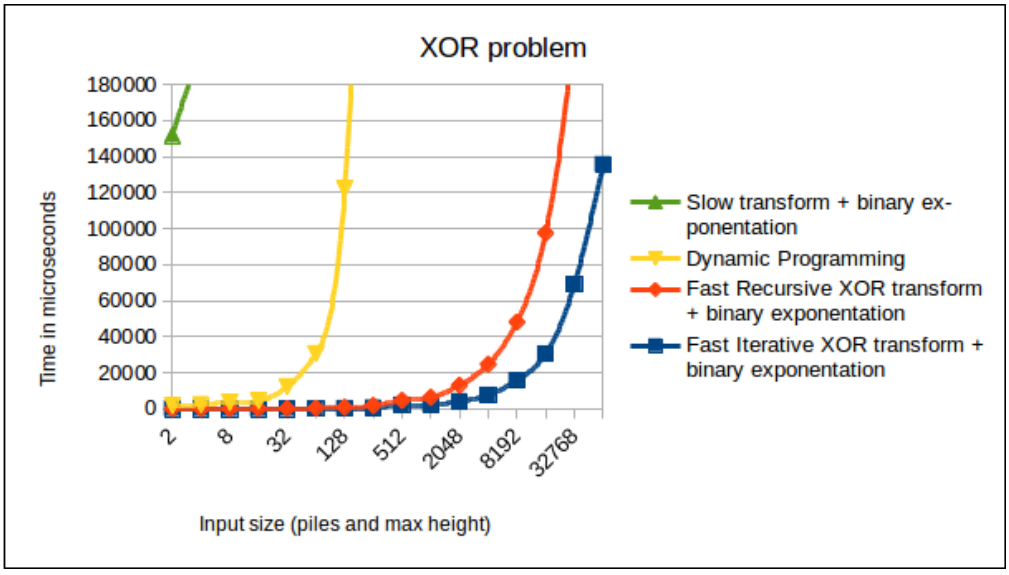


Figure 5.1: Prime Nim solutions performance

We can see that the iterative version of the Fast Walsh-Hadamard transform is faster than its recursive version as n gets bigger. For the Dynamic Programming, it works fine for small input, but as soon as we hit larger values ($n > 1000$ and $h > 1000$), time and memory exceeded the imposed restrictions. Finally, the importance of the binary exponentiation optimization is clearly seen in the graph: the Walsh-Hadamard transform solution without it grows way too fast, even worse than the Dynamic Programming solution.

Setting the constraints that the solution should support $n \leq 10^9$ and $h \leq 10^5$ and run in less than 3 seconds, both the Iterative and Recursive Fast-Walsh Hadamard transform with the binary exponentiation would be the only solutions to pass this criteria.

5.2 Results for problem MAXOR

Unlike the XOR problem, there is a solution for the OR problem that does not use convolutions at all and runs surprisingly fast. Below are the solution sketches for each of them:

- Fast OR transform: The key observation lied in creating two identical characteristic vectors, where 1 is at position i if i occurs in the array, and 0 otherwise. Then, performing OR convolution between these two vectors, and using basic combinatorics to compute the number of pairs for the highest non-zero element in the convolution vector was sufficient to obtain the answer. The time complexity of this solution is $\mathcal{O}(n \log n)$

- Dynamic Programming: This is a solution that is very particular to this problem, and runs surprisingly fast. Let's use a technique called Dynamic Programming Sum over Subsets (DP SOS). The idea behind DP SOS is that we can calculate the next DP state: Let $dp[mask]$ be the number of numbers in the set whose binary representation is a subset of the binary representation of $mask$. Setting $B = 17$ as the highest bit ON, and n the number of elements in the set, the array dp is calculated as shown in the next pseudocode:

for $i = 0$ to n **do**

  $dp[i] = 1$

for $b = 0$ to B **do**

  for $mask = 0$ to 2^B **do**

    if $mask \wedge 2^b$ **then**

      $dp[mask] \leftarrow dp[mask] + dp[mask \oplus 2^b]$

After computation of dp, we proceed to update $dp[mask]$ with $dp[mask] = \frac{dp[mask] * dp[mask-1]}{2}$. This essentially is counting the number of pairs such that their resulting OR is a submask of $mask$. But we would like only to only count the number of pairs whose OR is exactly $mask$. In order to do this, we apply DP SOS again:

---

for $i = 0$ to n **do**

  $dp[i] = 1$

for $b = 0$ to B **do**

  for $mask = 0$ to 2^B **do**

    if $mask \wedge 2^b$ **then**

      $dp[mask] \leftarrow dp[mask] - dp[mask \oplus 2^b]$

It can be noticed that the only difference is that we are subtracting instead of adding. For a given $mask$, subtraction is done to eliminate those pairs whose OR is never equal to $mask$, and those pairs can be taken from every submask of $mask$ that differ in one bit. As we are running this from lower values of $mask$ to higher values, all pairs that should not be counted will be considered. The time complexity of this solution is $\mathcal{O}(B2^B)$Brute force: Given n is the length of the array, and A is the array, do the following: For

every $i \in [0, n-1]$, and for fixed i, for every $j \in [i+1, n-1]$, compute their OR value, and check if it is bigger than the biggest OR value that has been recorded. If it is, update the biggest OR value and set the number of pairs equal to one. In case the OR value is equal to the highest recorded OR value, increment the number of pairs by one, and finally, if the OR value is less than the highest recorded OR value, do nothing. The time complexity of this solution is $\mathcal{O}(n^2)$

The next graph shows how each of the algorithms perform for different input sizes. The input sizes were given as powers of two (2, 4, 8, ..., 2^{16}) as shown in the x-axis, while the y-axis shows the time in microseconds each solution took to finish.

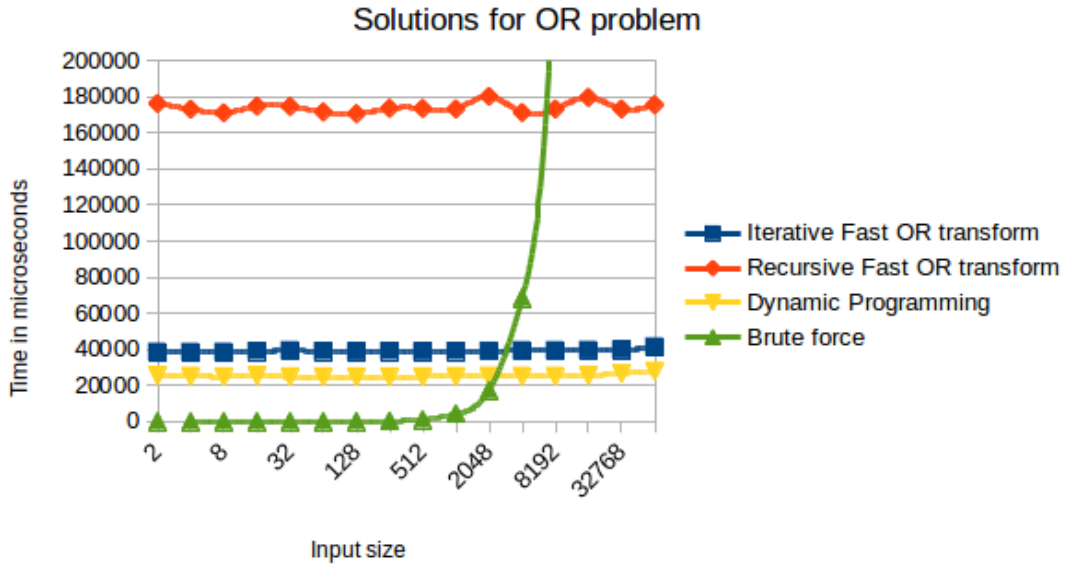


Figure 5.2: MAXOR solutions performance

It is surprising to see how the recursive solution starts with slightly big time values even for very small n values. Since a characteristic vector of size 10^5 is built, because the 10^5 depends on the biggest value that any element in the array may have, there is no way avoiding its creation and initialization regardless of n. Nevertheless, as n grows, the solution oscillates between the time range [160000, 180000] (in microseconds), since in reality n is irrelevant and we are always solving

the problem for the biggest value in the array. Hypothetically, if we could iterate from 1 to infinite in no time, but the biggest value in the array was 10^5, the solution would run in the same time. Otherwise, if $n = 1$ but the biggest value were infinite, the solution would never finish.

The case with the brute force solution is the opposite: we might have huge values in the array, and that would not be a problem timewise (except for overflow), because n dictates the number of operations we will be performing. Trying every pair of values for small n is reasonable, but then our time function starts growing very fast as soon as $n > 2^{16}$.

5.3 Results for problem AND closure

Just like the OR problem, it is possible to come up with a clever Dynamic Programming solution that has a very good runtime. Actually, there is another knapsack-like Dynamic Programming solution, very similar to that from the XOR results part, but unfortunately much worse. It won't be discussed as the superior Dynamic Programming solution will take place. Without further due, the solution sketches are as follows:

- Fast AND transform: Once again, a characteristic vector of the input array A was required. The key point to notice in order to understand that AND-convolution correctly solves this problem is that, at any given step, performing bitwise AND between two values, one of which already contained the other, is the same as skipping that value. If it was not contained, then the bitwise AND between both values will take it into account and possibly create a new AND value. Skipping a value is equivalent to not consider it in a given subset, and taking it into account means it belongs to this subset. As every subset is generated by either considering or not every element of the array, it is the case that this is indeed what we would like to do. The number of times we can consider or not a value is the number of elements in the array, that is n. Applying the binary exponentiation optimization, we do not have to consider or ignore one by one, but instead consider values in chunks. The number of chunks can be at most $\log n$. As AND-convolution is done at every step, the time complexity of this solution is $\mathcal{O}n \log^2 n$

- Dynamic Programming: First, let's notice that the restrictions imposed in the problem mention that the biggest value that can occur in the array is up to 10^6. The closest power of two that is bigger to this value is 20, which means every number has at most 20 bits on. When we take the bitwise AND of two values, we never obtain a bigger value: it either stays the same or decreases. Since every number has at most 20 bits, the bitwise AND of every subset will have at most 20 bits as well. That being said, we would like to know, for every number x from 0 to 2^{20}, if there is a subset of numbers S from the array such that their bitwise AND is x.
  How to tell whether the bitwise AND of a subset S of numbers is equal to x? The following has to be true:

  - For every bit b that is ON in x, every $y \in S$ must have that bit ON.
  - For every bit b that is OFF in x, at least one $y \in S$ must have that bit OFF. If none of the elements in S had bit b OFF, then x would not be their bitwise AND value.

  Then, for a fixed x, if we are able to find such subset S satisfying the previous conditions, then x can be obtained as the bitwise AND of the elements of S. There is a clever way to compute this:
  First, initialize array $dp[i] = 2^{20} - 1$ for every $i \in [0, 2^{20} - 1]$. Then, for each value v that occurs in the array, set $dp[v] = v$. Since it was mentioned that the bitwise AND operation always results in a smaller or equal value, we can perform computations in decreasing order of possible AND value x, that is, x goes from $2^{20} - 1$ to 0. Then the next pseudocode solves the problem:

```
answer ← 0
for x = 2^20 − 1 downto 0 do
  if dp[x] == x then
    answer ← answer + 1
  for b = 0 to 20 do
    if x ∧ 2^b then
      dp[x ⊕ 2^b] ← dp[x ⊕ 2^b] ∧ dp[x]
```

The answer is incremented by one when $dp[x] = x$, because it means that the bitwise AND of every value in the array that contained x as a submask resulted in the bits that are OFF in x to be OFF in such set of values as well. For the purpose of this solution, that means that it is possible to obtain x from a subset of values. But we had to identify every value v that contained x as a submask. To avoid iterating over every value and checking if $x \wedge v = x$, we instead remove one bit that is ON from v, and the resulting number w is a submask of v. As x is going down, eventually we will reach w, turn one bit OFF, get some value z which is both a submask of w (differing by one bit) and v (differing by two bits), and later x will be at z. By the time x is at where it is, all values who contain x in its submask are already considered, so then the value of $dp[x]$ can be checked to see if it was possible to obtain x. The time complexity of this solution is $\mathcal{O}(B2^B)$, where $B = 20$.Brute force: This one is about generating every subset, taking the bitwise AND of the elements in the subset, and adding the resulting number to a data structure that supports counting each number once, such as a binary search tree. At the end, the answer is the number of elements in the data structure (in this case, the number of vertices in the binary search tree). If insertion to the data structure takes time T, the complexity of this solution is $\mathcal{O}(nT2^n)$. For binary search trees, $T = \mathcal{O}(\log n)$.

Finally, the next graph shows how each of the solutions performed against increase inputs of size 2^n for $n \in [1, 16]$ as shown in the x-axis. The y-axis shows

the time in microseconds each solution took to finish.

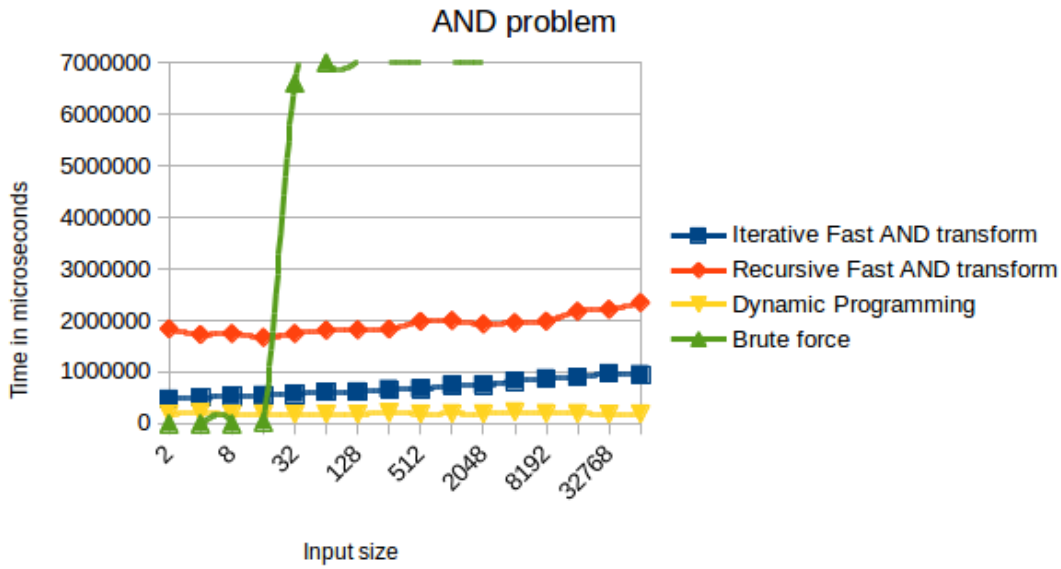


Figure 5.3: AND closure solutions performance

Since we are trying every subset, as soon as we try more than 2^{20} subsets, we can tell that it is going to take a long time to finish. Once again, the iterative version of the fast AND transform beats the recursive version, but the Dynamic Programming solution overtakes them all.

Chapter 6

Conclusions

The XOR, OR and AND convolutions turned out to be very versatile tools for problem solving. A variety of problems that can be expressed in terms of logical convolutions can easily sped up by applying the corresponding operation. For two of the three problems discussed, there were intended solutions which did not use at all these convolutions, but coming up with such solutions is totally dependent on the problem. For instance, both MAXOR and AND closure were optimal in performance by using Dynamic Programming, but since Dynamic Programming is more of a paradigm rather than a direct tool, there are multiple ways to formulate solutions in terms of recurrences. After going to the solution sketches for the Dynamic Programming solutions, it can be noticed that they are completely different; it is up to the creativity of the problem solver to be able to find the best solution under this paradigm. On the other hand, solving the problems put emphasis on discovering a formulation in terms of convolutions, which then led to a straightforward applications of them.

Beyond using these convolutions as tools for problem solving, it was very surprising to the author that there was very little work on them. As noted in the State of the Art chapter, there is a lot of work on the transforms themselves, but very little discussion on how they can take advantage of their properties to use the convolution theorem and the sort of convolution they compute. Hopefully, this work can contribute a bit to the discussion of contributions. Most of the work out there on transforms and convolutions is directed to their applications in the sciences, but this one is targeted to a community of problem solvers that enjoy competition

and the study of algorithms. It can be also be said that this work is a survey of all of the resources out there in the Competitive Programming community related to convolutions that are spread throughout the web.

Bibliography

[1] About us - topcoder. URL: https://www.topcoder.com/about-topcoder/.

[2] Csacademy round 13 analysis. URL: https://csacademy.com/contest/round-13/analysis/.

[3] Csacademy round 53 task maxor. URL: https://csacademy.com/contest/round-53/task/maxor/.

[4] Fast fourier transform and its applications. *Real Time Digital Signal Processing*, page 303–350. doi:10.1002/0470845341.ch7.

[5] Fast fourier transform and variations of it. URL: https://csacademy.com/blog/fast-fourier-transform-and-variations-of-it.

[6] Fast walsh-hadamard transform (fwht) explanation (translated). URL: http://blog.csdn.net/xuanandting/article/details/70991372.

[7] General ideas. URL: http://codeforces.com/blog/entry/48417.

[8] Problem name: Nim. URL: https://community.topcoder.com/tc?module=ProblemDetail&rd=14543&pm=11469.

[9] Single round match 518 editorial. URL: https://apps.topcoder.com/wiki/display/tc/SRM518.

[10] JORG ARNDT. *MATTERS COMPUTATIONAL: ideas, algorithms, source code*. SPRINGER-VERLAG BERLIN AN, 2016.

[11] Jörg Arndt. Matters computational. 2011. doi:10.1007/978-3-642-14764-7.

[12] Sheldon Jay. Axler. *Linear algebra done right*. Springer, 1996.

[13] Thomas H.. Cormen, Charles Eric. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*.

[14] Alejandro Dominguez. A history of the convolution operation [retrospectroscope]. *IEEE Pulse*, 6(1):38–49, 2015. doi:10.1109/mpul.2014.2366903.

[15] Moshe Dror, Pierre LEcuyer, and Ferenc Szidarovszky. *Modeling Uncertainty: an Examination of Stochastic Theory, Methods, and Applications*. Springer Science Business Media, Inc., 2005.

[16] A.c. Elster. Fast bit-reversal algorithms. *International Conference on Acoustics, Speech, and Signal Processing*. doi:10.1109/icassp.1989.266624.

[17] Thomas S. Ferguson. *Game theory*. World Scientific, 2017.

[18] I. J. Good. Introduction to cooley and tukey (1965) an algorithm for the machine calculation of complex fourier series. *Springer Series in Statistics Breakthroughs in Statistics*, page 201–216, 1997. doi:10.1007/978-1-4612-0667-5_9.

[19] M. Heideman, D. Johnson, and C. Burrus. Gauss and the history of the fast fourier transform. *IEEE ASSP Magazine*, 1(4):14–21, 1984. doi:10.1109/massp.1984.1162257.

[20] Israel Kleiner. *A history of abstract algebra*. Birkhauser, 2007.

[21] Steven George. Krantz. *Handbook of complex variables*. Birkhauser, 1999.

[22] David K. Maslen and Daniel N. Rockmore. The cooley–tukey fft and group theory. *Journal of the American Mathematical Society*, 10(01):169–215, Jan 1997. doi:10.1090/s0894-0347-97-00219-1.

[23] G. Robinson. Logical convolution and discrete walsh and fourier power spectra. *IEEE Transactions on Audio and Electroacoustics*, 20(4):271–280, 1972. doi:10.1109/tau.1972.1162394.

[24] Fatiul Huq Sujoy. Competitive programming - where computer geeks become gladiators, May 2016. URL: https://www.thedailystar.net/shout/cover-story/competitive-programming-where-computer-geeks-become-gladiators-1222453.

[25] vexorian. Tco 2012 round 2a - topcoder wiki. URL: https://apps.topcoder.com/wiki/display/tc/TCO2012Round2A.

[26] J. L. Walsh. A closed set of normal orthogonal functions. *Joseph L. Walsh*, page 109–128, 2000. doi:10.1007/978-1-4612-2114-2_11.

[27] J. E Jr. Whelchel and D. F. Guinn. *The Fast Fourier-Hadamard Transform and Its Use in Signal Representation and Classification.* Defense Technical Information Center, 1968.

[28] Michal Wos. Walsh-hadamard transform and its application in linearity testing of boolean functions. *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2009*, 2009. doi:10.1117/12.837769.

[29] Leonid P. Yaroslavsky. Fast transforms in image processing: Compression, restoration, and resampling. *Advances in Electrical Engineering*, 2014:1–23, 2014. doi:10.1155/2014/276241.