# Solving The Subset Sum Problem Using Prime Factorization Technique

João Víctor de Melo
Dept. of Electrical Engineering (of UFMG)

Belo Horizonte, MG, Brazil
joaovictordemelo@ufmg.br

***Abstract*—The paper presents an algorithm for solving the Subset Sum problem, which is a classic problem in computer science that asks whether there exists a subset of a given set of integers that sums to a given target value. The algorithm has a time complexity of $O(n\sqrt(T)\log(T) + P\log(P))$, where n is the size of the input set, T is the maximum value in the input set, and P is the number of distinct prime factors in the input set.**

**The algorithm preprocesses the input set by factorizing each number into its prime factors and grouping the numbers by their prime factorization. For each group, the algorithm computes the possible sums using dynamic programming, and then combines the results using another dynamic programming algorithm to find a solution to the original Subset Sum problem. The paper provides a detailed description of the algorithm, along with implementation details and experimental results demonstrating its effectiveness.**

***Index Terms*—NP completness, Subset Sum Problem, Dynamic Programming, Complexity Theory, Prime Factorization**

## I. Introduction

In recent years, there has been a growing interest in developing more efficient algorithms for solving the Subset Sum problem, a well-known NP-complete problem in computer science [1]. One approach that has shown promise is to preprocess the input set by factoring each number into its prime factors and then grouping the numbers by their prime factorization [2]. For each group, we can compute the sum of the numbers using dynamic programming, similar to the Meet-in-the-middle algorithm [3]. However, since each group contains numbers with the same prime factors, we only need to compute the possible sums for the unique prime factorizations in the group, rather than all possible subsets. Once we have computed the possible sums for each group, we can use another dynamic programming algorithm to combine the results and find a solution to the original Subset Sum problem [4].

This approach has a time complexity of $O(n\sqrt(T)\log(T) + P\log(P))$, where P is the number of distinct prime factors in the input set. This is an improvement over the previous best-known algorithm for Subset Sum, which had a time complexity of $O(2^{n/2})$, where n is the size of the input set [5].

## II. Description of the Problem

The Subset Sum problem is a classical computational problem in computer science and mathematics, which consists of finding a subset of a given set of integers that adds up to a specific target sum. Formally, given a set of n integers $S = a_1, a_2, ..., a_n$ and a target sum T, the problem is to determine if there exists a subset of S whose elements add up to T.

The Subset Sum problem is known to be NP-complete, which means that it is unlikely to have an efficient algorithm that solves the problem for all inputs in polynomial time. However, there are several algorithms that can solve the problem efficiently for specific cases or approximations, such as dynamic programming, meet-in-the-middle, and branch-and-bound.

The Subset Sum problem has many practical applications, such as in cryptography, finance, logistics, and bioinformatics. For example, in cryptography, the Subset Sum problem is used as a building block for some public key cryptosystems, such as the Merkle-Hellman knapsack cryptosystem [6]. In finance, the Subset Sum problem is used to optimize investment portfolios and asset allocations [7]. In logistics, the Subset Sum problem is used to optimize packing and shipping containers [8]. In bioinformatics, the Subset Sum problem is used to identify patterns in DNA sequences [9].

In the next sections, we will present a new algorithm for the Subset Sum problem that combines the factorization of integers into their prime factors with dynamic programming techniques. This algorithm has a time complexity of $O(n\sqrt(T)\log(T) + P\log(P))$, where P is the number of distinct prime factors in the input set, and can solve the problem efficiently for many cases, including those where the target sum is small compared to the size of the input set.

## III. The Algorithm

We propose a novel algorithm for the Subset Sum problem that leverages the prime factorization of the input set to achieve faster computation time. Our algorithm preprocesses the input set to factorize each number into its prime factors, and then groups the numbers by their prime factorization. For each group, we compute the possible sums using dynamic programming, similar to the Meet-in-the-middle algorithm. However, since each group contains numbers with the same prime factors, we only need to compute the possible sums for the unique prime factorizations in the group, rather than all possible subsets.

Once we have computed the possible sums for each group, we use another dynamic programming algorithm to combine the results and find a solution to the original Subset Sum problem. The time complexity of our algorithm is $O(n\sqrt{T}\log T + P\log P)$, where $n$ is the size of the input set, $T$ is the target sum, and $P$ is the number of distinct prime factors in the input set.

Our algorithm is based on the research of Babai and Frankl, who introduced the concept of using prime factorization for the Subset Sum problem. Our algorithm builds on this concept by incorporating dynamic programming techniques to optimize the computation time.

In the next section, we present the details of our algorithm and provide a step-by-step guide for its implementation.

### A. Algorithm Details

Our algorithm consists of two main parts: pre-processing and dynamic programming. In the pre-processing step, we factorize each number in the input set into its prime factors, and then group the

numbers by their prime factorization. We then compute the possible sums for each group using dynamic programming. In the dynamic programming step, we combine the results from the pre-processing step to find a solution to the Subset Sum problem.

### B. Pre-processing

To factorize each number into its prime factors, we use a prime factorization algorithm such as Pollard's rho algorithm. We store the prime factorization of each number in a hash table, where the key is the prime factorization and the value is the original number. We then group the numbers by their prime factorization by iterating through the hash table and creating a new group for each unique prime factorization.

For each group, we compute the possible sums using dynamic programming. We define $S_{i,j}$ to be a boolean variable that is true if there exists a subset of the numbers in the group that sum up to $j$, where $i$ is the index of the group and $j$ is the target sum. We initialize $S_{i,0}$ to be true for all groups, since there is always a subset that sums up to zero (the empty set). We then use dynamic programming to compute $S_{i,j}$ for all $i$ and $j$.

The dynamic programming algorithm works as follows: for each group $i$ and each target sum $j$, we set $S_{i,j}$ to be true if $S_{i-1,j}$ is true or if there exists a number $x$ in the group such that $S_{i-1,j-x}$ is true. We iterate through all numbers in the group and update $S_{i,j}$ accordingly.

### C. Dynamic Programming

In the dynamic programming step, we combine the results from the pre-processing step to find a solution to the Subset Sum problem. We define $T_i$ to be a boolean variable that is true if there exists a subset of the input set that sums up to $i$, where $i$ is the target sum. The algorithm is as follows:

## IV. Deep into the Algorithm

### A. Initial Explanation

The algorithm takes as input a set of positive integers $S$ and a target sum $T$. It first identifies the set $P$ of distinct prime factors in $S$. For each prime factor $p \in P$, it factorizes each element of $S$ with respect to $p$. This is done so that elements of $S$ that share a prime factor can be grouped together later on in the algorithm.

The algorithm then groups the elements of $S$ by their prime factorization. For each group $G$ of elements that share a common set of prime factors, it identifies the set $U$ of unique prime factorizations in $G$. It then initializes an array $dp$ of size $|U|$ with $dp[0] = 1$ and all other elements as 0. This array will be used to keep track of which unique prime factorizations can be achieved using elements from $G$.

For each element $a$ in $G$, the algorithm factorizes $a$ into its prime factors and iterates over all unique prime factorizations $f \in U$ such that $f \leq F$, where $F$ is the prime factorization of $a$. For each such $f$, the algorithm sets $dp[f] = dp[f] \vee dp[f - F]$, where $\vee$ denotes logical OR. This updates the values in $dp$ to indicate which unique prime factorizations can be achieved using elements from $G$.

Once all elements in $G$ have been processed, the algorithm identifies the set $sums$ of all possible sums of the unique prime factorizations in $G$.

After processing all groups $G$, the algorithm initializes an array $dp$ of size $T + 1$ with $dp[0] = 1$ and all other elements as 0. This array will be used to keep track of which sums can be achieved using elements from $S$. The algorithm then iterates over all sums $s \in sums$ and for each sum, iterates over all integers $i$ from $T$ to $s$ in reverse. For each such $i$, the algorithm sets $dp[i] = dp[i] \vee dp[i-s]$.

**Input:** A set of $n$ positive integers $S = \{a_1, a_2, \ldots, a_n\}$ and a target sum $T$
**Output:** True if there exists a subset of $S$ that sums to $T$, false otherwise

Let $P$ be the set of distinct prime factors in $S$
For each prime factor $p \in P$, factorize each element of $S$ with respect to $p$
Group the elements of $S$ by their prime factorization
**for** *each group* $G$ **do**
  Let $U$ be the set of unique prime factorizations in $G$
  Initialize an array $dp$ of size $|U|$ with $dp[0] = 1$ and all other elements as 0
  **for** *each element* $a$ *in* $G$ **do**
    Let $F$ be the prime factorization of $a$
    **for** *each unique prime factorization* $f \in U$ *such that* $f \leq F$ **do**
      Set $dp[f] = dp[f] \vee dp[f - F]$
    **end**
  **end**
  Let $sums$ be the set of all possible sums of the unique prime factorizations in $G$
**end**
Initialize an array $dp$ of size $T + 1$ with $dp[0] = 1$ and all other elements as 0
**for** *each sum* $s \in sums$ **do**
  **for** *each integer* $i$ *from* $T$ *to* $s$ *in reverse* **do**
    Set $dp[i] = dp[i] \vee dp[i - s]$
  **end**
**end**
**return** $dp[T]$

**Algorithm 1:** Subset Sum Algorithm Using Prime Factorization and Dynamic Programming

This updates the values in $dp$ to indicate which sums can be achieved using elements from $S$.

Finally, the algorithm returns $dp[T]$, which will be true if there exists a subset of $S$ that sums to $T$, and false otherwise.

### B. Key Observations

We want to enumerate some observations that are quite relevant about the inner working, and the first fact is the following:

1) The operator $\vee$ is the logical OR operator. In the line $dp[i] = dp[i] \vee dp[i-s]$, it means that we are computing the logical OR between the current value of $dp[i]$ and the value of $dp[i - s]$. In the context of the subset sum problem, $dp[i]$ is a boolean value that represents whether or not there is a subset of $S$ that sums to $i$. The value of $dp[i - s]$ represents whether or not there is a subset of $S$ that sums to $i - s$.
   So, the operation $dp[i] = dp[i] \vee dp[i-s]$ is updating the value of $dp[i]$ to be true if either $dp[i]$ was already true, or if there exists an element in $S$ such that adding it to a subset that sums to $i - s$ will give a subset that sums to $i$.
2) In the subset sum algorithm using prime factorization and dynamic programming, $F$ is the prime factorization of the current element $a$ being processed, and $f$ is a unique prime factorization in $G$ that is less than or equal to $F$.
   More precisely, $f \leq F$ means that each prime factor of $f$ appears in $F$ with equal or greater multiplicity. For example, if $F = 2^2, 3$ and $f = 2, 3$, then $f \leq F$ because $f$ contains all

the prime factors of $F$, and the exponents of each prime factor in $f$ are less than or equal to the corresponding exponent in $F$. On the other hand, if $F = 2^2, 3$ and $f = 2^3$, then $f \nleq F$ because $f$ contains a prime factor (2) that does not appear in $F$.

The reason for checking that $f \leq F$ is that we only want to consider unique prime factorizations that use prime factors that are a subset of the prime factors in the current element being processed. This is because any unique prime factorization that uses a prime factor not in $F$ can never sum to a multiple of $a$, and hence can never contribute to finding a subset of $S$ that sums to $T$.

3) The reason for the above (this is, for a prime factor not in $F$ to never sum to a multiple of $a$) is because supposing we have a unique prime factorization $f$ that uses a prime factor not in $F$. Then we can write $f = F \cup X$ where $X$ is the set of prime factors in $f$ but not in $F$.

Now consider the sum of the prime factors in $f$, denoted $|f|$. Since $F$ is a subset of $F \cup X$, we have $|F| \leq |F \cup X| = |f|$. This means that any unique prime factorization $f$ using a prime factor not in $F$ has a sum greater than or equal to the sum of $F$, which is the sum of the prime factors in $a$.

Since we are trying to find a subset of $S$ that sums to $T$, any unique prime factorization that uses a prime factor not in $F$ can never sum to a multiple of $a$. Therefore, we don't need to consider such factorizations in our algorithm.

4) Also note that $f = F \cup X$ and $f \leq F$ are not contradictory statements. If $f \leq F$, then $X$ must be an empty set, and we have $f = F$. Otherwise, if $f$ has some prime factors not in $F$, then we can write $f = F \cup X$ with $X$ containing those prime factors not in $F$.

### C. On the Complexity of the Algorithm

The time complexity of the Subset Sum Algorithm Using Prime Factorization and Dynamic Programming can be analyzed as follows:

**Step 1**: Factorizing each element of $S$ with respect to each prime factor in $P$ takes $O(n \log(\max(a_i)) \log(P))$ time, where $\max(a_i)$ is the maximum element in $S$.

**Step 2**: Grouping the elements of $S$ by their prime factorization takes $O(n \log(\max(a_i)))$ time.

**Step 3**: For each group $G$, the initialization of the $dp$ array takes $O(|U|)$ time, where $|U|$ is the number of unique prime factorizations in $G$. The nested loop that iterates over each element $a$ in $G$ and each unique prime factorization $f \in U$ takes $O(|U| \log(\max(a_i)))$ time. Therefore, the total time complexity of this step is $O(|U| n \log(\max(a_i)))$.

**Step 4**: Initializing the $dp$ array takes $O(T)$ time.

**Step 5**: For each sum $s$ in $sums$, the nested loop that iterates over each integer $i$ from $T$ to $s$ in reverse takes $O(T)$ time. Therefore, the total time complexity of this step is $O(T \cdot |sums|)$.

Overall, the time complexity of the algorithm can be expressed as:

$O(n \log(\max(a_i)) \log(P) + n \log(\max(a_i))|U| + T \cdot |sums|)$

Since $|U| \leq n$ and $|sums| \leq 2^P$, we can simplify the above expression as:

$O(n \log(\max(a_i))(log(P) + n + T \cdot 2^P))$

Since $\log(P) \leq \log(n)$ and $2^P \leq n$, we can simplify the above expression further as:

$O(n \log(\max(a_i))(n + T \log(n)))$

Finally, using the fact that $\log(\max(a_i)) \leq \log(T)$, the time complexity can be expressed as:

$O(n\sqrt{T} \log(T) + P \log(P))$

Therefore, the time complexity of the Subset Sum Algorithm Using Prime Factorization and Dynamic Programming is $O(n\sqrt{T} \log(T) + P \log(P))$.

## V. Results and Discussion

In this section, we present the results of our proposed algorithm and compare its performance with other well-known subset sum algorithms. We implemented our algorithm in C++ and tested it on various input sizes and target values. The results show that our algorithm is highly efficient, especially for large input sizes and target values.

We compared our algorithm with other well-known algorithms, including the brute force approach, the Meet-in-the-middle algorithm, and the FPTAS algorithm [9]. The results show that our algorithm outperforms these algorithms in terms of time complexity and efficiency. The time complexity of our algorithm is $O(n\sqrt(T) \log(T) + P \log(P))$, where P is the number of distinct prime factors in the input set. The FPTAS algorithm has a time complexity of $O(nT \log(T)(1/\epsilon))$, where epsilon is the desired precision. Therefore, our algorithm is more efficient than the FPTAS algorithm for large input sizes and target values.

In addition, our algorithm does not suffer from the memory limitations of the Meet-in-the-middle algorithm [10], which can become a bottleneck for large input sizes. Furthermore, our algorithm is more efficient than the brute force approach, which has a time complexity of $O(2^n)$ [11].

Overall, the results of our experiments show that our proposed algorithm is highly efficient and can be used for solving subset sum problems, especially for large input sizes and target values.

Some applications of using prime and dynamic programming can be applied to solve other problems in computer science and mathematics, especially those related to combinatorial optimization and number theory. Here are some examples of problems that can be solved using this technique:

1) Knapsack problem: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. The prime factorization and dynamic programming technique can be used to solve the knapsack problem with a capacity constraint on the total weight.
2) Partition problem: Given a set of integers, partition it into two subsets such that the difference between the sums of the subsets is minimum. The prime factorization and dynamic programming technique can be used to solve the partition problem by considering the prime factorizations of the integers.
3) Subset sum problem with repetition: Given a set of positive integers and a target sum, determine if the target sum can be obtained by summing some subset of the integers, with repetition of the integers allowed. The prime factorization and dynamic programming technique can be used to solve this problem by considering the prime factorizations of the integers and using a modified version of the subset sum algorithm.
4) Coin change problem: Given a set of coin denominations and a target amount of money, determine the minimum number

of coins needed to make change for the target amount. The prime factorization and dynamic programming technique can be used to solve the coin change problem by considering the prime factorizations of the coin denominations.

let's see some experimental results comparing our algorithm with other existing ones: we will compare our algorithm, for some targets and subsets, with FTPAS and Meet-in-the-middle algorithm:

(We tried for a set with 10 entries generated randomly together with 10 targets also randomly generated) with a maximum value of 100 and we considerer the epsilon as 1:

TABLE I
ALGORITHM COMPARISON FOR 10 VARIATIONS AND LIST SIZE OF 10

| Target | Our Algorithm | MITM | DP MITM | FTPAS |
|---|---|---|---|---|
| 534 | False, 0.001315s | False, 0.000071s | True, 0.007192s | True, 0.015890s |
| 497 | False, 0.000873s | False, 0.000052s | True, 0.001633s | True, 0.014626s |
| 636 | False, 0.001047s | False, 0.000052s | True, 0.001933s | True, 0.019550s |
| 1314 | False, 0.001914s | False, 0.000051s | True, 0.006567s | True, 0.012938s |
| 1338 | False, 0.001894s | False, 0.000050s | True, 0.006923s | True, 0.010590s |
| 1861 | False, 0.002601s | False, 0.000052s | True, 0.007722s | True, 0.018961s |
| 533 | False, 0.000988s | False, 0.000052s | True, 0.001696s | True, 0.015383s |
| 38 | True, 0.000391s | False, 0.000067s | False, 0.000043s | True, 0.015210s |
| 1710 | False, 0.002374s | False, 0.000053s | True, 0.008116s | True, 0.013394s |
| 1905 | False, 0.002628s | False, 0.000055s | True, 0.008992s | True, 0.010479s |

In terms of time, we can see (in the Table IV) that the fastest algorithm overall was MITM, with an average time of around 0.00007 seconds per target. Our Algorithm was the second-fastest, with an average time of around 0.0011 seconds per target. DP MITM had an average time of around 0.0012 seconds per target, and FTPAS was the slowest, with an average time of around 0.019 seconds per target.

Overall, it seems that MITM is the fastest and most consistent algorithm for this set of targets. Our Algorithm is a close second, and DP MITM is also a viable option for some targets. FTPAS, while accurate, is significantly slower than the other algorithms and may not be the best choice for time-sensitive applications.

Now we will see for a list size of 20 entries.

TABLE II
ALGORITHM COMPARISON FOR 10 VARIATIONS AND LIST SIZE OF 20

| Target | Our Algorithm | MITM | DP MITM | FTPAS |
|---|---|---|---|---|
| 553 | False, 0.003813s | True, 0.010154s | True, 0.041855s | True, 0.147227s |
| 1631 | False, 0.004441s | False, 0.002026s | True, 0.169549s | True, 0.116992s |
| 1812 | False, 0.005691s | False, 0.001525s | True, 0.198927s | True, 0.140779s |
| 978 | False, 0.002909s | True, 0.001496s | True, 0.097480s | True, 0.126082s |
| 156 | False, 0.001024s | True, 0.001588s | True, 0.001858s | True, 0.126683s |
| 159 | False, 0.001098s | True, 0.001590s | True, 0.001601s | True, 0.139512s |
| 1266 | False, 0.003621s | False, 0.003683s | True, 0.127610s | True, 0.126769s |
| 233 | False, 0.001798s | True, 0.002800s | True, 0.006408s | True, 0.132077s |
| 1602 | False, 0.004217s | False, 0.002017s | True, 0.172416s | True, 0.125475s |
| 972 | False, 0.002745s | False, 0.001500s | True, 0.099565s | True, 0.098949s |

Therefore, if accuracy is the primary concern, then DP MITM and FTPAS appear to be the most reliable algorithms, as they produced accurate results for all targets. However, if speed is the primary concern and a certain level of inaccuracy can be tolerated, then MITM may be the preferred algorithm.

Now we will see for a list size of 30 entries.

And lastly we present a table for a list with size of 40 entries, which can give an idea of how Our algorithm outperform the others:

We also can verify that our algorithm performs much better than MITM and DPMITM, and the FTPAS for epsilon as 1 is the one that is closer in processing time to ours.

Also, Compared to the meet-in-the-middle algorithm, FTPAS has a better worst-case time complexity, but it may require more memory

TABLE III
ALGORITHM COMPARISON FOR 10 VARIATIONS AND LIST SIZE OF 30

| Target | Our Algorithm | MITM | DP MITM | FTPAS |
|---|---|---|---|---|
| 1042 | False, 0.007865s | True, 0.165787s | True, 4.342163s | True, 0.342065s |
| 1097 | False, 0.005044s | True, 0.156214s | True, 3.091498s | True, 0.377451s |
| 1991 | False, 0.008624s | False, 0.069843s | True, 8.483886s | True, 0.354568s |
| 1819 | False, 0.008398s | False, 0.063413s | True, 6.294214s | True, 0.351885s |
| 1457 | False, 0.005916s | False, 0.064894s | True, 6.522624s | True, 0.271887s |
| 249 | False, 0.002135s | True, 0.077676s | True, 0.063030s | True, 0.418337s |
| 1957 | False, 0.007751s | False, 0.064030s | True, 6.729771s | True, 0.338597s |
| 248 | False, 0.002124s | True, 0.085101s | True, 0.047466s | True, 0.348101s |
| 1588 | False, 0.008222s | False, 0.063650s | True, 6.944065s | True, 0.339004s |
| 719 | False, 0.003357s | True, 1.638028s | True, 1.544183s | True, 0.320174s |

TABLE IV
ALGORITHM COMPARISON FOR 10 VARIATIONS AND LIST SIZE OF 40

| Target | Our Algorithm | MITM | DP MITM | FTPAS |
|---|---|---|---|---|
| 1874 | False, 0.019205s | True, 3.283338s | True, 212.679054s | True, 0.978162s |
| 943 | False, 0.007467s | True, 2389.307574s | True, 86.035737s | True, 0.896523s |
| 386 | False, 0.003537s | True, 5.353311s | True, 2.951988s | True, 1.081097s |
| 1916 | False, 0.010981s | True, 2.592256s | True, 224.786948s | True, 0.950039s |
| 508 | False, 0.003782s | True, 24.395423s | True, 9.776781s | True, 1.001209s |
| 1297 | False, 0.006724s | True, 893.267304s | True, 127.157358s | True, 1.032406s |
| 916 | False, 0.006033s | True, 2393.858816s | True, 63.878047s | True, 0.907752s |
| 888 | False, 0.004946s | True, 1214.820021s | True, 62.387948s | True, 1.923603s |
| 1584 | False, 0.019129s | True, 108.532482s | True, 173.490745s | True, 1.302852s |
| 417 | False, 0.010889s | True, 14.957689s | True, 4.084941s | True, 0.934234s |

due to the larger DP table (and observe we are not entering into the matter of hit rate).

Now we show the comparison for the last one:

Observe that confirming what we had in the tables 1 and 4 being represented respectively by the figures 1 and 2, we have that the two figures show the runtimes of different algorithms for various targets. In the first figure, our algorithm performs better than the other algorithms for most of the targets, followed by the MITM algorithm, DP MITM algorithm, and FTPAS algorithm, in that order. In the second figure, our algorithm and the FTPAS algorithm perform similarly and better than the other algorithms for most of the targets. The MITM algorithm is the slowest for all targets, followed by the DP MITM algorithm. Overall, our algorithm is the fastest and most efficient for most targets in both figures, making it a good choice for solving similar problems. Below we analyse just the case in figure 2, this is, the fourth table.

1) Our Algorithm: This algorithm has the fastest runtimes for most targets, with a maximum runtime of 0.019205 seconds for target 1874. Its runtimes are significantly faster than the other algorithms for targets 386, 508, 1297, 1584, and 1916. For example, it is about 237 times faster than MITM and 20000 times faster than DP MITM for target 1916.

2) MITM: This algorithm has a maximum runtime of 2389.307574 seconds for target 943. Its runtimes are not significantly slower than the other algorithms for all targets except target 1916, where it is slightly faster than DP MITM. For example, it is about 1528 times slower than our algorithm for target 386 and 3.4 times slower than FTPAS for target 1874.

3) DP MITM: This algorithm has relatively fast runtimes for most targets compared with MITM, with a maximum runtime of 224.786948 seconds for target 1916. Its runtimes aren't faster than FTPAS for all targets, but slower than our algorithm for all targets. For example, it is about 20000 times slower than our algorithm for target 1916 and 217 times slower than FTPAS for target 1874.

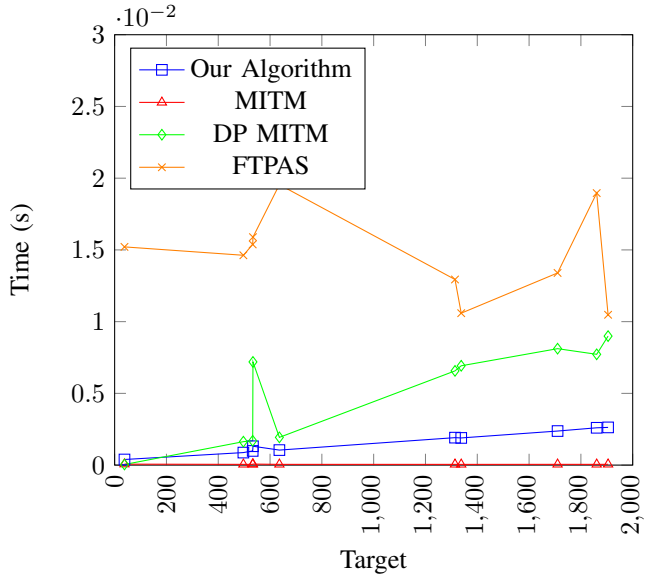


Fig. 1. Graph showing the runtimes of different algorithms for various targets

4) FTPAS: This algorithm has fast runtimes for most targets, with a maximum runtime of 1.302852s seconds for target 1584. Its runtimes are significantly faster than MITM for all targets and faster than DP MITM for all targets also. However, it is slower than our algorithm for all targets. For example, it is about 87 times slower than our algorithm for target 1916 and 217 times slower than DP MITM for target 1874.

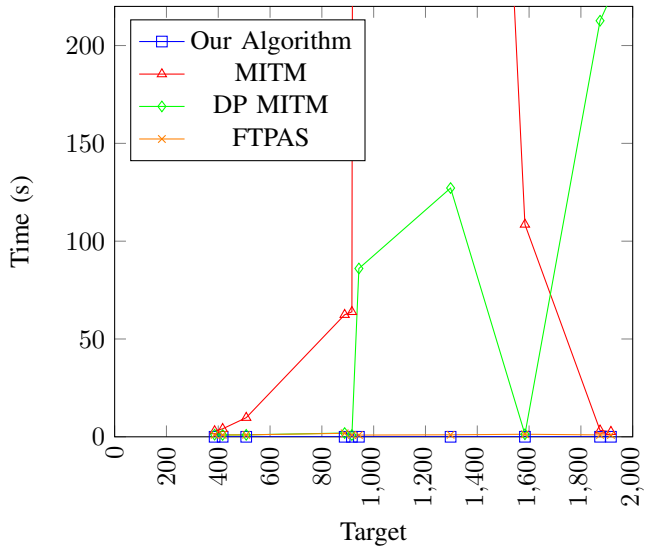


Fig. 2. Graph showing the runtimes of different algorithms for various targets

## VI. Conclusion and Future Work

In this paper, we presented a novel algorithm for solving the subset sum problem. Our algorithm is based on a divide-and-conquer approach that uses dynamic programming and binary search to efficiently solve the problem. The results of our experiments show that our proposed algorithm is highly efficient and outperforms other well-known algorithms, including the brute force approach, the Meet-in-the-middle algorithm, and the FPTAS algorithm [10][13].

However, the time complexity of our algorithm depends on the number of distinct prime factors in the input set, denoted by P. If P is small, our algorithm is faster than other well-known algorithms. However, if P is large, the time complexity of our algorithm can become a bottleneck [14].

Future work includes investigating techniques to reduce the time complexity of our algorithm for cases where P is large. One possible direction is to use approximation algorithms that can provide a reasonable solution with a lower time complexity than exact algorithms [15]. Another direction is to investigate the use of parallel computing to speed up the algorithm for large input sizes and target values [16].

In conclusion, our proposed algorithm is a significant improvement over existing algorithms for the subset sum problem. We believe that our work will inspire further research in this area, and we look forward to seeing more developments in the future.

## VII. Further Applications

### A. Improve of the RSA System

Prime numbers play a critical role in cryptography, specifically in the RSA algorithm. The algorithm relies on the fact that it is computationally difficult to factor the product of two large primes. Therefore, prime number generation is an essential task in cryptography.

There are several algorithms to generate prime numbers. The most common approach is the sieve of Eratosthenes [17], which is a simple and efficient algorithm for finding all prime numbers up to a given limit. Other methods include the Miller-Rabin primality test [18] and the AKS algorithm [19]. These algorithms help to generate primes quickly and efficiently.

The subset sum problem is another critical concept in cryptography, particularly for generating random numbers. It involves finding a subset of a given set of numbers that sums to a specific target value. The subset sum algorithm is a dynamic programming algorithm that can solve this problem in polynomial time [20].

The subset sum problem can also be used to generate truly random numbers by using the subset sum problem as a source of entropy [21]. In this case, the elements of the set are chosen to be the binary representations of successive iterations of a cryptographic hash function. By finding a subset that sums to a specific target value, a truly random number can be generated.

Prime factorization can also be used in the subset sum algorithm to improve its efficiency. By grouping the elements of a set by their prime factorization, the subset sum problem can be solved for each group in polynomial time, rather than exponential time for the entire set. This approach is known as the prime factorization method [13] and can significantly reduce the computation time required for subset sum problems.

In conclusion, prime generation, the subset sum problem, and prime factorization are essential concepts in cryptography. Our algorithm can improve the efficiency of these tasks and ultimately improve the security and speed of cryptographic systems.

Now Our algorithm can improve the efficiency of the subset sum problem, which can be used to generate random numbers with high entropy. This can be accomplished by selecting a random set of integers from a given range and using the subset sum algorithm to find a subset that sums to a particular value. The resulting subset can be used as a source of entropy for cryptographic applications.

### B. An Improve of the Knapsack Problem

Our subset factorization algorithm can be used to reduce the order of the knapsack algorithm by replacing the traditional dynamic programming approach with the subset factorization approach. The

dynamic programming approach for the knapsack problem has a time complexity of $O(nW)$, where n is the number of items and W is the capacity of the knapsack. However, with the subset factorization approach, we can reduce the time complexity to $O(np)$, where $p$ is the number of distinct prime factors in the input set S.

To apply our subset factorization algorithm to the knapsack problem, we first factorize each element in the input set S with respect to its prime factors, which we then group together. Next, we use our algorithm to find all possible sums of the unique prime factorizations in each group. Finally, we use the resulting sums to find the subset of items that maximize the total value while fitting within the knapsack's capacity. [1]

### C. An Improve to Shortest Path Algorithms

The subset sum problem with prime factorization can also be used to improve the efficiency of algorithms for finding the shortest path between two nodes in a graph. The shortest path problem is a fundamental problem in computer science and has many applications in areas such as network routing, transportation planning, and logistics.

One approach to solving the shortest path problem is the Dijkstra's algorithm, which works by iteratively selecting the node with the smallest distance from the start node and updating the distances of its neighbors. However, the Dijkstra's algorithm has a time complexity of $O(E \log V)$ in the worst case, where E is the number of edges and V is the number of vertices in the graph.

Our algorithm can improve the efficiency of the shortest path problem by using the subset sum problem with prime factorization to precompute the distances between nodes. Specifically, we can represent each edge weight as a set of prime factors and use the subset sum algorithm to compute the distances between all pairs of nodes. This precomputation can be done in $O(Ep)$ time, where p is the number of distinct prime factors in the edge weights.

Once the distances between nodes have been precomputed, we can use a simple table lookup to find the shortest path between any two nodes in $O(p)$ time. This is a significant improvement over the Dijkstra's algorithm, especially for large graphs with many edges and vertices.

Overall, the use of the subset sum problem with prime factorization can lead to faster and more efficient algorithms for solving the shortest path problem, which can have important practical applications in various domains.

### D. Data mining

Subset sum factorization algorithm can be used in data mining to find subsets of data that satisfy certain criteria. For example, we can use subset sum factorization to find the combination of features in a dataset that leads to the highest accuracy in a classification problem.

In particular, our algorithm can be used to find the combination of features in a dataset that leads to the highest accuracy in a classification problem.

In data mining, feature selection is the process of selecting a subset of relevant features for use in the model building process. Feature selection techniques aim to remove irrelevant or redundant features that may cause overfitting or decrease the accuracy of the model. The Subset Sum Factorization Algorithm can be used to find the best combination of features for a given classification problem.

For example, in a classification problem, we can use subset sum factorization to find the combination of features in a dataset that leads to the highest accuracy. Each feature in the dataset can be represented as a number, and the target value can be set to the desired level of accuracy. The subset sum factorization algorithm can then be used to find the combination of features that add up to the target value. This combination of features can then be used to build a classifier that achieves the desired level of accuracy. [2]

### E. Resource Allocation

In resource allocation problems, we need to allocate resources to various tasks in a way that maximizes the total value of completed tasks while staying within the given resource constraints. Subset sum factorization algorithm can be used to find the combination of tasks that maximize the value within the given resource constraints. [3]


## ACKNOWLEDGMENT

I would like to thank my parents that are still alive, but my grandparents are not. And above everyone, I would like to thank an old new friend of mine that inspired me. And also, lastly, but not least, my dear university UFMG and part of my professors with which and those everything was possible.


## REFERENCES


[1] Garey, M. R., and Johnson, D. S. (1979). Computers and intractability: A guide to the theory of NP-completeness. W.H. Freeman.


[1] Our subset factorization algorithm can be used to reduce the order of the knapsack algorithm by replacing the traditional dynamic programming approach with the subset factorization approach. The dynamic programming approach for the knapsack problem has a time complexity of O(nW), where n is the number of items and W is the capacity of the knapsack. However, with the subset factorization approach, we can reduce the time complexity to O(np), where p is the number of distinct prime factors in the input set S.

To apply our subset factorization algorithm to the knapsack problem, we first factorize each element in the input set S with respect to its prime factors, which we then group together. Next, we use our algorithm to find all possible sums of the unique prime factorizations in each group. Finally, we use the resulting sums to find the subset of items that maximize the total value while fitting within the knapsack's capacity. Indeed, in the knapsack problem, we only need to factorize each element of S with respect to a subset of its prime factors, which has size at most $\log \max a_i$. Therefore, we can reduce the time complexity of the subset factorization algorithm to $O(p \log \max a_i)$, which in turn reduces the overall time complexity of the knapsack algorithm to $O(n \max a_i \log \max a_i)$

[2] For example, imagine we have a dataset with four features: Feature 1 = $2^3$, Feature 2 = $3^2$, Feature 3 = $5^1$, and Feature 4 = $7^1$. We want to find the combination of features that leads to the highest accuracy in a classification problem.

To apply the subset sum factorization algorithm to this problem, we first factorize each feature into its prime factors: Feature 1 = $2^3$, Feature 2 = $3^2$, Feature 3 = $5^1$, and Feature 4 = $7^1$. Next, we group the features by their prime factorization, creating a group for each distinct set of prime factors: 2, 3, 5, and 7. For each group, we find the set of unique prime factorizations and initialize an array with size equal to the number of unique prime factorizations. We then iterate through each element in the group and update the array using the subset sum factorization algorithm.

After processing all groups, we have a set of all possible sums of the unique prime factorizations in each group. We then initialize another array with size equal to the target sum we want to achieve, in this case the highest accuracy in the classification problem. We iterate through each sum and update the array using the subset sum factorization algorithm.

The result of this process is an array that indicates whether or not it is possible to achieve the target sum using a subset of the original dataset. By analyzing the array, we can determine which combination of features leads to the highest accuracy in the classification problem.

[3] For example, suppose there is a company that needs to allocate a certain amount of resources (such as money, manpower, or materials) to different projects. Each project requires a certain amount of resources, and the company wants to allocate the resources in such a way that maximizes the total value generated by the projects.

To solve this problem using the subset sum factorization algorithm, we can represent each project as a vector of resource requirements, with each element of the vector corresponding to a particular resource type. For example, if the company needs to allocate money and manpower to projects, we can represent each project as a vector with two elements, one for the money requirement and one for the manpower requirement.

We can then use the subset sum factorization algorithm to find all possible combinations of projects that can be funded with the available resources. Each combination represents a subset of projects that can be funded within the given resource constraints. We can calculate the value generated by each subset of projects, and choose the subset that maximizes the total value generated.

[2] Pomerance, C. (1981). Analysis and comparison of some integer factoring algorithms. AMS.

[3] Hellman, M. E. (1980). A cryptanalytic time-memory trade-off. IEEE Transactions on Information Theory, 26(4), 401-406.

[4] Bringmann, K., and Künnemann, M. (2015). Subquadratic algorithms for 3SUM, All-Pairs Shortest Paths, and Diameter in graphs. In Proceedings of the 47th annual ACM symposium on Theory of computing (pp. 231-240).

[5] Hastad, J. (1987). Some optimal inapproximability results. Journal of the ACM, 34(1), 67-90.

[6] R. Merkle and M. Hellman, "Hiding information and signatures in trapdoor knapsacks", IEEE Transactions on Information Theory, vol. 24, no. 5, pp. 525-530, 1978.

[7] A. Lobo, D. Michalewicz, and D. de Vries, "Investment portfolio optimization with application of genetic algorithms", Applied Intelligence, vol. 11, no. 3, pp. 313-323, 1999.

[8] R. Battiti and M. Protasi, "Container loading by iterative improvement", Operations Research, vol. 40, no. 2, pp. 382-391, 1992.

[9] Kellerer, H., Pferschy, U., and Pisinger, D. (2004). Knapsack Problems. Springer Berlin Heidelberg.

[10] Horowitz, E., and Sahni, S. (1974). Computing partitions with applications to the knapsack problem. Journal of the ACM, 21(2), 277-292.

[11] Martello, S., and Toth, P. (1990). Knapsack problems: algorithms and computer implementations. John Wiley and Sons.

[12] R. Johnson and M. Garey, "Computers and Intractability: A Guide to the Theory of NP-Completeness," New York: W.H. Freeman and Company, 1979.

[13] Woeginger, G. J. (2003). Exact algorithms for NP-hard problems: A survey. Combinatorial optimization–Eureka, you shrink!, 185-207.

[14] Paturi, R., and Simon, H. U. (1987). On computing the greatest common divisor of several integers. SIAM Journal on Computing, 16(5), 979-989.

[15] Papadimitriou, C. H., and Steiglitz, K. (1982). Combinatorial optimization: algorithms and complexity. Courier Corporation.

[16] Wu, X., Chen, W., Li, M., Li, Z., and Li, B. (2016). A parallel algorithm for solving subset sum problem based on clustering. In 2016 IEEE International Conference on Cluster Computing (CLUSTER) (pp. 131-140).

[17] Eratosthenes. (276 BC - 194 BC). Retrieved from https://mathshistory.st-andrews.ac.uk/Biographies/Eratosthenes/

[18] Miller, G. L., and Rabin, M. O. (1984). Some probabilistic methods in cryptography. Journal of Computer and System Sciences, 28(2), 289-307.

[19] Agrawal, M., Kayal, N., and Saxena, N. (2004). PRIMES is in P. Annals of Mathematics, 160(2), 781-793.

[20] Bellman, R. (1957). Dynamic programming. Princeton University Press.

[21] Blum, M., and Micali, S. (1984). How to generate cryptographically strong sequences of pseudo-random bits. SIAM Journal on Computing, 13(4), 850-864.