

# A PROJECT REPORT

on

---

## *NP Completeness of Subset Sum Problem*

---

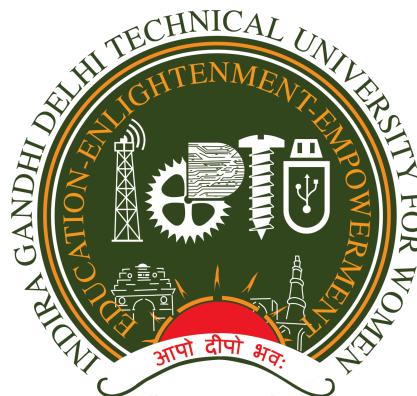
*Submitted in the requirements for the lab project for the course of B.Tech in CSE*  
**Subject: Design and Analysis of Algorithms Lab (BCS- 204)**

*Submitted By:*

**Ananya Arora - 02601012022**  
**Anavi Srivastava - 02801012022**  
**Anwita Sinha - 03901012022**  
**Arya Singh – 04401012022**  
**Bhavya Gupta - 05101012022**

*Under the guidance of*

**Dr. Vivekanand Jha**  
Associate Professor, CSE



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**  
**INDIRA GANDHI DELHI TECHNICAL UNIVERSITY FOR WOMEN**  
**KASHMERE GATE, DELHI-110006**

April 2024

## **TABLE OF CONTENTS**

<b>S.No.</b>	<b>Topic</b>	<b>Page No.</b>
1.	Acknowledgment	3
2.	Undertaking under Anti Plagiarism	4
3.	Certificate	5
4.	List of Tables	6
5.	List of Figures	7
6.	List of Equations	8

# **ACKNOWLEDGMENTS**

We extend our sincere thanks to Indira Gandhi Delhi Technical University for Women (IGDTUW) for providing us with the necessary platform and resources for our research on the Subset Sum Problem.

We express our deep gratitude to Assistant Professor, Dr. Vivekanand Jha for his unwavering support, guidance, and invaluable insights throughout our research journey. His commitment to fostering a conducive learning environment has significantly contributed to our academic progress.

Our appreciation also goes to the Computer Science and Engineering (CSE) department for their support and resources, which have been instrumental in our research endeavours.

We are grateful to our families and friends for their unwavering support and encouragement during this research undertaking.

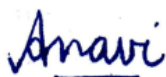
Special thanks to our team members for their collaboration, dedication, and collective efforts in ensuring the success of this research project.

## **UNDERTAKING REGARDING ANTI-PLAGIARISM**

We, *Ananya Arora, Anavi Srivastava, Anwita Sinha, Arya Singh* and *Bhavya Gupta*, hereby, declare that the material/ content presented in the project report is free from plagiarism and is properly cited and written in our own words. We are fully aware about the UGC regulations for promotion of academic integrity and prevention of plagiarism in higher educational institutions. In case plagiarism is detected at any stage, we shall be solely responsible for it.



Name :Ananya Arora  
Enrolment No: 02601012022



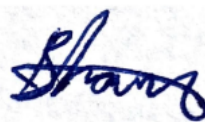
Name :Anavi Srivastava  
Enrolment No: 02801012022



Name :Arya Singh  
Enrolment No: 04401012022



Name :Anwita Sinha  
Enrolment No: 03901012022



Name :Bhavya Gupta  
Enrolment No: 05101012022

## **CERTIFICATE**

This is to certify that the project entitled ***NP Completeness of Subset Sum Problem*** is an authentic record of our own work, under the supervision of **Dr. Vivekanand Jha**, Associate Professor, CSE, Indira Gandhi Delhi Technical University for Women, Delhi. It is submitted in the requirements for the lab project of Design and Analysis of Algorithms (BCS- 204) for the course of the Bachelor of Technology degree in Computer Science and Engineering at Indira Gandhi Delhi Technical University for Women, Delhi, India. We further certify that:

- This work has not been submitted to any other Institution for any sort of degree/diploma/certificate in India or abroad.
- We totally abided by the plagiarism guidelines given to us by the university for writing and preparing this report.
- Whenever we have used materials (text, data, theoretical analysis/equations, codes/program, figures, tables, pictures, text etc.) from other sources, we have done proper citation, linking of the owner's rightful source and giving due credit wherever needed.



Name :Ananya Arora  
Enrolment No: 02601012022



Name :Arya Singh  
Enrolment No: 04401012022



Name :Anavi Srivastava  
Enrolment No: 02801012022



Name :Bhavya Gupta  
Enrolment No: 05101012022



Name :Anwita Sinha  
Enrolment No: 03901012022

---

**Dr. Vivekanand Jha**  
**Associate Professor, CSE**

## **LIST OF TABLES**

<b>Table No.</b>	<b>Section No.</b>	<b>Title</b>	<b>Page No.</b>
1.	3.3.3	Average Runtimes	26
2.	3.4.3	Strict Templates	29
3.	5.1.0	Comparison of Algorithms wrt Time and Space Complexity	61

## **LIST OF FIGURES**

<b>Figure No.</b>	<b>Section No.</b>	<b>Title</b>	<b>Page No.</b>
1.	3.2.4	A min-heap Binary tree	23
2.	3.2.5	A heap ordered 4-subset tree	24
3.	5.3.0	Runtime vs Array Size Graph for Algorithm#1	63
4.	5.3.0	Runtime vs Array Size Graph for Algorithm#2	63
5.	5.3.0	Runtime vs Array Size Graph for Algorithm#3	63
6.	5.3.0	Runtime vs Array Size Graph for Algorithm#4	63
7.	5.3.0	Runtime vs Array Size Graph for Algorithm#5	63
8.	5.4.0	Performance Comparison Graph for all Algorithms	64

## **LIST OF EQUATIONS**

<b>Equation No.</b>	<b>Section No.</b>	<b>Title</b>	<b>Page No.</b>
1.	2.2.1	Intuition behind proof of NP Complete	14
2.	2.2.4	Exactly 1 3-SAT $\leq_p$ M	16
3.	2.2.5	M $\leq_p$ SUBSET-SUM	17
4.	3.3.2	Introduction to Review of Paper#2	26
5.	3.3.4	Method outcome in Review of Paper#3	27
6.	4.2.2	Initialisation of CardPlayer's Algorithm	41-42
7.	4.2.3	Branch and Bound Algorithm Equations	50
8.	4.2.4	Parallel Algorithm Equations	53



# **INDEX**

<b>S.No.</b>	<b>Contents</b>	<b>Page No.</b>
1.	<b>Introduction</b> 1.1 NP completeness 1.2 P, NP, NPC and NPH	10 11
2.	<b>Problem statements</b> 2.1 Introduction 2.2 Proof of NP complete problem 2.3 Application of Subset Sum	12-13 14-18 19
3.	<b>Literature Review</b> 3.1 Review of Common Paper 3.2 Review of Paper#1 3.3 Review of Paper#2 3.4 Review of Paper#3 3.5 Review of Paper#4 3.6 Review of Paper#5	20-21 22-25 26-27 28-29 30 31
4.	<b>Implementation and Results</b> 4.1 Implementation environment 4.2 Individual paper results 4.2.1 Paper#1 4.2.1 Paper#2 4.2.1 Paper#3 4.2.1 Paper#4 4.2.1 Paper#5 4.3 Comparative results	32 33-40 41-49 50-52 53-55 56-50 61-64
5.	<b>Conclusion and Future Scope</b>	65
6.	<b>References</b>	66

# 1. INTRODUCTION

## 1.1 NP COMPLETENESS

In 1971, the STOC conference ignited a fervent debate over whether NP-complete problems could be efficiently tackled by deterministic Turing machines within polynomial time. Amidst this discussion, John Hopcroft played a pivotal role in steering the conference towards a consensus. Recognizing the absence of concrete proofs backing either assertion, he coined the term "the question of whether  $P=NP$ " and advocated for its postponement until further evidence emerged.

Decades later, the query remains one of mathematics' most tantalising puzzles, encapsulating the essence of the P versus NP problem. To incentivize resolution, the Clay Mathematics Institute has dangled a US\$1 million reward for a formal proof in either direction.

The bedrock of NP-completeness was laid by the Cook-Levin theorem in 1971, spotlighting the Boolean satisfiability problem. Richard Karp's subsequent work in 1972 expanded this foundation by unveiling several additional NP-complete problems. Garey & Johnson's seminal work in 1979 further amplified the roster of NP-complete challenges through reductions from established cases.

Formally, a problem earns the NP-complete label if it meets specific criteria. It must be a decision problem yielding a binary outcome, with a concise polynomial-length solution available when the answer is affirmative. Moreover, solutions must be swiftly verifiable, allowing for efficient validation in polynomial time or via exhaustive brute-force search. Crucially, NP-complete problems possess a universal simulation capability, signifying their role as benchmarks for the hardest problems with quickly verifiable solutions.

The term "NP-complete" derives from its components: "nondeterministic polynomial-time complete." This nomenclature underscores the role of nondeterministic Turing machines, where "polynomial time" denotes a computationally manageable duration for solution validation. Each input to an NP-complete problem associates with a set of solutions, swiftly verifiable in polynomial time, situating them within the NP complexity class.

Despite the swift verification of NP-complete solutions, uncovering them remains an elusive endeavour. As problem size escalates, so does the computational effort required, reinforcing the enigmatic nature of the P versus NP problem.

While a definitive solution methodology remains elusive, practitioners frequently encounter NP-complete challenges in various domains. Addressing these hurdles often entails employing heuristic methods and approximation algorithms, underscoring the ongoing quest to navigate the intricate landscape of computational complexity. <sup>[15][16]</sup>

## 1.2 P, NP, NPC and NP-Hard Problems

Computational complexity theory serves as a beacon in the vast ocean of computer science, illuminating the pathways to understanding the resources needed to tackle computational problems. It delves into the intricate dance between problem size and the resources—such as time and space—required for their solution.

At the heart of this theory lie four key classifications: P, NP, NP-complete, and NP-hard, each representing a distinct facet of computational complexity.

1. **P (Polynomial Time):** In the realm of P, problems are akin to well-behaved citizens—solutions to them can be swiftly unearthed using algorithms with polynomial time complexity. Jack Edmonds laid the groundwork for this class in 1965, with Stephen Cook and Leonid Levin later refining the concept in the 1970s. Problems in P, such as sorting and searching, are considered tractable, making them the bread and butter of algorithmic problem-solving.
2. **NP (Nondeterministic Polynomial Time):** Enter the enigmatic world of NP, where solutions to problems may be elusive, but their correctness can be efficiently verified. Stephen Cook introduced NP in 1971, unveiling its complexity through the lens of the Boolean satisfiability problem. NP problems, while seemingly challenging to solve directly, hold a tantalising promise—if a polynomial-time algorithm were to crack one, it would imply the resolution of the infamous P vs. NP conjecture, a Holy Grail of computer science.
3. **NPC (NP-Complete):** The crown jewel of NP, NP-complete problems reign supreme as the most formidable adversaries in computational complexity. Stephen Cook's seminal work in 1971 unveiled the concept of NP-completeness, showcasing problems like the Boolean satisfiability problem as the gatekeepers of computational complexity. If one were to devise a polynomial-time solution for any NP-complete problem, the entire landscape of NP problems would fall like dominos, revealing a unified realm of efficiently solvable problems.
4. **NP-Hard (Nondeterministic Polynomial-Time Hard):** But amidst the hierarchy of complexity lies a wild terrain inhabited by NP-hard problems—problems that are as hard as, if not harder than, the toughest problems in NP. Alan Turing's exploration into the decision version of the Halting Problem in 1936 laid the foundation for this classification, showcasing problems that transcend the boundaries of NP yet pose formidable challenges for algorithmic conquest. NP-hard problems, while not necessarily verifiable in polynomial time, serve as benchmarks for computational difficulty, pushing the boundaries of what algorithms can achieve.

In conclusion, the classifications of P, NP, NP-complete, and NP-hard serve as guideposts in the labyrinth of computational complexity theory. They illuminate the landscape of problem-solving challenges, offering insights into the capabilities and limitations of algorithms and computers. As we navigate this intricate terrain, we unravel the mysteries of computation, inching closer to unlocking the true potential of problem-solving in the digital age. <sup>[17]</sup>

## 2. PROBLEM STATEMENT

### 2.1 INTRODUCTION

The Subset Sum Problem is a fundamental concept in computer science that falls under the category of NP-completeness. The Subset Sum Problem is a significant challenge in fields such as complexity theory, bin packing, and cryptography. In this report, we aim to review six research papers based on the Subset Sum problem. We have explored different solutions/algorithms to reduce its time complexity from exponential i.e in the order of  $O(2^N)$  to polynomial time complexity  $O(N^x)$  that make use of various algorithms and data structures. We subsequently perform a comparative analysis on these algorithms to observe their performance based on different parameters.

#### 2.1.1 PROBLEM DEFINITION:

Given a set of non-negative integers  $S = \{a_1, a_2, \dots, a_n\}$  and a target sum  $K$ , the Subset Sum Problem asks whether there exists a **subset** of numbers from  $S$  that **add up exactly** to  $K$ .

##### Example:

- Let  $S = \{3, 7, 4\}$  and  $K = 10$ .

The problem asks if there's a subset of numbers in  $S$  that adds up to 10. In this case, yes,  $\{3, 7\}$  is a subset that sums to 10.

The Subset Sum Problem's role in NP-completeness is crucial for understanding the complexity of computational problems.

#### 2.1.2 NP (NON-DETERMINISTIC POLYNOMIAL):

A problem is considered in NP if a solution (in this case, a subset that sums to  $K$ ) can be verified in polynomial time. For Subset Sum, verifying a solution is straightforward. We simply add all the elements in the claimed subset and compare the sum to  $K$ . This verification process takes time proportional to the number of elements in the subset, which is always less than or equal to the total number of elements in the original set ( $n$ ). Since verification time grows polynomially with the input size ( $n$ ), Subset Sum qualifies for the NP category.

#### 2.1.3 NP-HARD:

A problem is NP-hard if any problem in the NP class can be reduced to it in polynomial time. Reduction here means transforming an instance of another NP problem into an instance of the Subset Sum Problem, while preserving the solution. The key point is that this reduction must be achieved efficiently, meaning the transformation takes polynomial time.

Here's why Subset Sum is considered NP-hard:

- Several other NP problems can be reduced to Subset Sum in polynomial time. For example, the Partition Problem (dividing a set into two subsets with equal sums) can be reduced to Subset Sum by checking if half the total sum exists in a subset.

- If we had an efficient (polynomial time) algorithm for Subset Sum, we could solve any problem in NP efficiently. This is because we could reduce any NP problem to Subset Sum and then use the efficient Subset Sum algorithm to find the solution for the original NP problem.

#### **2.1.4 NP-COMPLETENESS:**

A problem is NP-complete if it satisfies both being in NP (verifiable in polynomial time) and NP-hard (any NP problem can be reduced to it efficiently). Subset Sum fulfils both these conditions, making it NP-complete.

#### **Significance of NP-Completeness:**

NP-completeness helps categorise problems based on their computational difficulty. If a problem is NP-complete, it's highly likely that there's no efficient (polynomial time) algorithm for solving it for all input sizes. This doesn't necessarily mean it's impossible to solve, but it suggests that finding an optimal solution for large instances might be computationally expensive.

Knowing that a problem is NP-complete guides researchers in their approach to solving it. They may focus on:

- Approximation algorithms: These algorithms find solutions that are close to optimal (within a guaranteed percentage) but have polynomial time complexity.
- Heuristics: These are problem-specific techniques that often provide good solutions but may not guarantee optimality.
- Efficient algorithms for specific cases: While there might not be a universally efficient algorithm for Subset Sum, efficient solutions might exist for specific scenarios (e.g., sets with small sizes or limited range). <sup>[11]</sup>

## 2.2 PROOF OF NP COMPLETE PROBLEM

In the SUBSET SUM problem, we are given a list of  $n$  numbers  $A_1, \dots, A_n$  and a number  $T$  and need to decide whether there exists a subset  $S \subseteq [n]$  such that

$$\sum_{i \in S} A_i = T$$

### 2.2.1. INTUITION

We are reducing 3-SAT to EXACTLY 1 3-SAT which is further reduced to  $M$  (defined below). This  $M$  can be reduced to SUBSET-SUM problems.  $M$  is defined as the problem of finding whether a solution exists for a set of  $k$  equations  $E_1$  to  $E_k$ . Each equation  $E_i$  is of the form

$$a_{i1}x_1 + a_{i2}x_2 + \dots a_{in}x_n = b$$

where  $b = [0 \ 0 \ 0 \dots 0 \ 1]$   $a_{ij} = [0 \ 0 \ 0 \dots 0 \ 0]$  or  $[0 \ 0 \ 0 \dots 0 \ 1]$ .

Each of the variables  $x_i$  can only take the values 0 or 1.  
 $i \in \{1, 2, \dots, k\}$  and  $j \in \{1, 2, \dots, n\}$

$$3\text{-SAT} \leq_p \text{EXACTLY 1 3-SAT} \leq_p M \leq_p \text{SUBSET-SUM}$$

### 2.2.2. SUBSET-SUM is NP

#### Solution polynomial sized

A solution of the subset-sum problem is a set  $S$  of indices  $i$  from the set  $[n]$ , which correspond to the  $A_i$ s which sum up to the number  $T$ . So the solution size could be at most  $n$ . So it is polynomial in the input size.

#### Polynomial time verification

Once we have the set  $S$ , we can verify the solution by summing up the corresponding  $A_i$ s and comparing this sum with  $T$ . The number of additions is at most  $n-1$ . So the addition and comparison can be done in polynomial time. Hence, SUBSET-SUM is in NP.

### 2.2.3. CLAIM 1 : 3-SAT p EXACTLY 1 3-SAT

In the EXACTLY ONE 3SAT problem, we are given a 3CNF formula  $\Phi$  and need to decide if there exists a satisfying assignment  $u$  for  $\Phi$  such that every clause of  $\Phi$  has exactly one TRUE literal.

Suppose  $\Psi$  is a 3-SAT expression. Therefore, it will be of the form

$$\Psi = C_1 \wedge C_2 \wedge \dots \wedge C_k$$

where each clause

$$C_i = (x \vee y \vee z)$$

where  $i \in \{1, 2, \dots, k\}$

We would convert the expression  $\Psi$  to an EXACTLY ONE 3SAT expression  $\Phi$  by making the following changes for every clause of  $\Psi$ : For every clause  $C_i = (x \vee y \vee z)$  in  $\Psi$ , introduce 6 new variables  $a_x, b_x, a_y, b_y, a_z, b_z$  and form the equivalent clause

$$D_i = (\neg x \vee a_x \vee b_x) \wedge (\neg y \vee a_y \vee b_y) \wedge (\neg z \vee a_z \vee b_z) \wedge (a_x \vee a_y \vee a_z)$$

**Claim :**  $C_i \equiv D_i$

**Proof :** For any particular assignment which satisfies  $C_i$ , we will choose our additional literals

$(a_x, b_x, a_y, b_y, a_z, b_z)$  in such a way that exactly one of  $(a_x, a_y, a_z)$  will become true. Suppose  $\omega$  is false, then the clause  $(\neg \omega \vee a_\omega \vee b_\omega)$  becomes true automatically and so  $a_\omega$  and  $b_\omega$  are chosen to be false. If  $\omega$  is true, then either one of  $a_\omega$  or  $b_\omega$  are made true depending on whether any other of  $a_i$ s are true, as we want the last clause of  $D_i$  to be true. Also, we cannot have  $x, y$  and  $z$  all false as  $C_i$  is satisfied, so we don't need to consider the case where  $a_x, a_y$  and  $a_z$  all are false at the same time.

On the other hand, if the clause  $C_i$  is false, then all of  $x, y, z$  have to be false. This would mean that  $a_x, a_y$  and  $a_z$  are all false and so the last clause of  $D_i$  will become false and hence  $D_i$  will not be satisfied. Hence, when  $C_i$  has a satisfying assignment so does  $D_i$  and when the former does not have a solution, the latter is also false.

Also, since we are adding only 6 extra literals for each clause, so each  $D_i$  will be constructed in constant time, and therefore the whole reduction will take only polynomial time.

## 2.2.4. CLAIM 2 : EXACTLY 1 3-SAT $\leq_p$ M

Suppose  $\Phi$  is a 3-SAT expression. Therefore, it will be of the form

$$\Phi = C_1 \wedge C_2 \wedge \dots \wedge C_k$$

where each clause

$$C_i = (x \vee y \vee z)$$

where  $i \in \{1, 2, \dots, k\}$

We convert an instance of this problem into an instance of the problem M by the following steps: For each clause  $C_i$ , we will introduce an equation  $E_i$  of the form

$$a_{i1}x_1 + a'_{i1}x'_1 + a_{i2}x_2 + a'_{i2}x'_2 + \dots + a_{in}x_n + a'_{in}x'_n = b$$

where  $n$  = total number of variables in  $\phi$

$x_j$  = boolean variable corresponding to the Exactly1 3-SAT literal  $x_j$

$x'_j$  = boolean variable corresponding to the Exactly1 3-SAT literal  $\neg x_j$

$a_{ij}$  take values  $[0 \ 0 \ \dots \ 0 \ 0]$  if  $x_j$  is not present in the clause  $C_i$

$[0 \ 0 \ \dots \ 0 \ 1]$  if  $x_j$  is present in the clause  $C_i$

$a'_{ij}$  take values  $[0 \ 0 \ \dots \ 0 \ 0]$  if  $\neg x_j$  is not present in the clause  $C_i$

$[0 \ 0 \ \dots \ 0 \ 1]$  if  $\neg x_j$  is present in the clause  $C_i$

$b = [0 \ 0 \ \dots \ 0 \ 1]$

length of all above binary numbers is  $\lceil \log(2n) + 1 \rceil$

If there exists a solution to  $\Phi$ , we get values of all the variables  $(x_1, x_2, \dots, x_n)$  from which we can get the corresponding values for M's variables  $(x_1, x_2, \dots, x_n, x'_0, x'_1, \dots, x'_n)$ . Since  $\Phi$  is satisfied, each of its clauses is satisfied. If we consider the equation  $E_i$  corresponding to the clause  $C_i$ , then the former would be true as of all the variables in the equation whose coefficient is a non-zero binary number (which means they are present in the clause) exactly one will be having the value 1, so the equation holds.

If there does not exist a satisfying solution to  $\phi$ , then there must be at least a single clause with a false value, so all literals in that clause will be false and hence the equation corresponding to that clause will not hold.

During reduction, the computations involved are: 1. Checking whether a literal exists in a particular clause (to find the value of  $a_{ij}$ ) takes constant time per literal. 2. Addition of  $n$  binary numbers for checking whether a solution satisfies the equation could take  $O(n \log(n))$  steps. So, computations can be done in polynomial time.

Hence proved.



### 2.2.5. CLAIM 3 : $M \leq_p$ SUBSET-SUM

$M$  is defined as the problem of finding whether a solution exists for a set of  $k$  equations  $E_1$  to  $E_k$ . Each equation  $E_i$  is of the form

$$a_{i1}x_1 + a_{i2}x_2 + \dots a_{in}x_n = b$$

where  $b = [0 \ 0 \ 0 \ \dots 0 \ 1]$   $a_{ij} = [0 \ 0 \ 0 \ \dots 0 \ 0]$  or  $[0 \ 0 \ 0 \ \dots 0 \ 1]$ .

Each of the variables  $x_j$  can only take the values 0 or 1.

$i \in \{1, 2, \dots, k\}$  and  $j \in \{1, 2, \dots, n\}$

For every instance of the problem  $M$ , we create an instance of the SubsetSum problem in the following way:

Define  $n$  numbers  $A_1, A_2, \dots, A_n$  such that  $A_j = a_{1j}, a_{2j}, \dots, a_{ij}, \dots, a_{kj}$ .

Also, define the number  $T = bbb \dots b$  ( $k$  times)

If the given set of equations have a solution, then form a subset  $S$  such that  $j \in S$  if  $x_j = 1$  in the satisfying solution. If  $x_{m_1}, x_{m_2}, \dots, x_{m_l}$  take value 1 in the satisfying solution ( $l \leq n$ ), and  $m_1, m_2, \dots, m_l \in \{1, 2, \dots, n\}$ , then

$$\sum_{t=m_1}^{m_l} a_{it} = b \quad \forall i \in \{1, 2, \dots, k\}$$

Hence,

$$\begin{aligned} \sum_{j \in S} A_j &= \left( \sum_{t=m_1}^{m_l} a_{1t} \right) \left( \sum_{t=m_1}^{m_l} a_{2t} \right) \dots \left( \sum_{t=m_1}^{m_l} a_{kt} \right) \\ &= bb \dots b(k \text{ times}) \\ &= T \end{aligned}$$

On the other hand, if there is no solution for the given set of equations, then for any assignment of  $(x_1, x_2, \dots, x_n)$ , at least one of the equations wouldn't be satisfied, i.e.

$$\sum_{t=m_1}^{m_l} a_{it} = c$$

for some  $i$ , and  $c \neq b$ , therefore,

$$\begin{aligned} \sum_{j \in S} A_j &= \left( \sum_{t=m_1}^{m_l} a_{1t} \right) \left( \sum_{t=m_1}^{m_l} a_{2t} \right) \dots \left( \sum_{t=m_1}^{m_l} a_{kt} \right) \\ &= bb \dots bcb \dots b(k \text{ times}) \\ &\neq T \end{aligned}$$

For every instance of the problem  $M$ , we can print the corresponding subset sum problem in  $O(k \cdot n \log(n))$  steps as  $T$ .

Also, during computation, in worst case, when we need to add all the  $A_j$ s, we would need  $O(n^2 \log(n))$  steps for total  $n-1$  additions.  
Hence proved.

### 2.2.6. CONCLUSION

SUBSET-SUM is in NP  $3\text{-SAT} \leq_p \text{EXACTLY } 1 \text{ } 3\text{-SAT} \leq_p M \leq_p \text{SUBSET-SUM}$   $3\text{-SAT}$  is NP-Complete.

Using the transitivity of reduction:

From Claim-1, EXACTLY 1 3-SAT becomes NP-Complete.

From Claim-2, M becomes NP-Complete.

And finally from Claim-3, SUBSET-SUM becomes NP-Complete.

Hence Proved. <sup>[18]</sup>

## 2.3 APPLICATION OF SUBSET SUM PROBLEM

The Subset Sum problem has various practical applications across different fields:

1. **Cryptanalysis:** It's used in cryptanalysis to break cryptographic schemes based on knapsack problems, such as the Merkle-Hellman knapsack cryptosystem.
2. **Finance:** In portfolio optimization, it's applied to find the best combination of financial assets that meet certain criteria, like risk tolerance or expected return.
3. **Data Mining:** It's utilised in data mining for pattern recognition, where subsets of data need to be analysed to identify trends or anomalies.
4. **Resource Allocation:** In resource allocation problems, such as scheduling tasks on machines or assigning resources to projects, Subset Sum helps optimise the allocation to meet constraints.
5. **Genomics:** In genomics, it's used to identify sequences of DNA that are associated with particular genetic traits or diseases.
6. **Telecommunications:** In telecommunications, it's applied to problems like channel assignment and frequency allocation in wireless communication networks.

These are just a few examples, but the Subset Sum problem has many other applications across various domains.

### 3. LITERATURE REVIEW

#### 3.1 REVIEW OF COMMON PAPER

Dynamic Programming for the Subset Sum Problem by Hiroshi Fujiwara, Hokuto Watari and Hiroaki Yamamoto.

##### 3.1.1 SUBSET SUM PROBLEM

The subset sum problem involves determining whether a set of integers can be combined to equal a specific target value.

##### 3.1.2 APPROACHES

###### 1. BRUTE FORCE APPROACH:

This method involves checking all possible combinations of integers in the set to see if any sum up to the target value. It's straightforward but can be very time-consuming for larger sets.

**Example:** Let's consider the subset sum problem with the set  $\{-8, -2, 5, 7, 9\}$  and the target sum of 10.

###### Explanation:

1. Individual Numbers: Initially, we check each number individually to see if any of them equals the target sum of 10. We start with -8, then -2, then 5, then 7, and finally 9. None of these numbers individually equals 10.
2. Pairs of Numbers: Since we didn't find a match with individual numbers, we move on to pairs of numbers. We check all possible pairs to see if their sum equals 10. The pairs we check are  $\{-8, -2\}$ ,  $\{-8, 5\}$ ,  $\{-8, 7\}$ ,  $\{-8, 9\}$ ,  $\{-2, 5\}$ ,  $\{-2, 7\}$ ,  $\{-2, 9\}$ ,  $\{5, 7\}$ ,  $\{5, 9\}$ ,  $\{7, 9\}$ . Among these pairs, only the pair  $\{-2, 5\}$  sums to 3, which is not equal to 10.
3. Triplets and Beyond: Since no pairs yield the target sum, we would continue to check triplets, quadruples, and so on. We check all possible triplets, quadruples, and larger combinations until we either find a combination that sums to 10 or exhaust all possibilities.

**Result:** In this example, after checking all possible combinations, we find that the triplet  $\{-2, 5, 7\}$  sums to 10, providing a valid solution to the subset sum problem for this set and target sum.

**Efficiency Analysis:** While this brute force approach guarantees finding a solution if one exists, it becomes inefficient for larger input sets. The computational complexity is  $O(2^N * N)$ , where  $N$  is the number of elements in the set. This means the time taken grows exponentially with the size of the input set, making it impractical for very large sets.

## 2. DYNAMIC PROGRAMMING SOLUTION:

Dynamic programming offers a solution to the subset sum problem in pseudo polynomial time complexity, denoted as  $O(sN)$ , where  $s$  is the target value and  $N$  is the length of the set.

### Algorithm Overview:

The dynamic programming approach maintains an array of Boolean values  $Q(N, s)$  to track the possibility of achieving each possible sum from 0 to the target value using elements from the set. The algorithm employs recursive arithmetic operations for each element in the set from the index of the target value.

### Initialization:

1. Initialise a 2D Boolean array  $Q$  of size  $(N+1) \times (s+1)$ , where  $N$  is the length of the set and  $s$  is the target sum.
2. Set  $Q[i][0] = \text{true}$  for all  $i$  from 0 to  $N$ , indicating that a subset with sum 0 can always be formed by choosing no elements.

### Dynamic Programming Steps:

1. Iterative Computation: Iterate through each element  $\text{num}$  in the set from index 1 to  $N$  and each possible sum  $j$  from 1 to  $s$ .
2. Update Boolean Values: For each element  $\text{num}$ , update  $Q[i][j]$  as  $Q[i-1][j]$  (if  $\text{num} > j$ ) or  $Q[i-1][j]$  OR  $Q[i-1][j-\text{num}]$  (if  $\text{num} \leq j$ ).
  - If  $\text{num} > j$ , it means including  $\text{num}$  would exceed the current sum  $j$ , so we inherit the possibility from the previous row.
  - If  $\text{num} \leq j$ , we consider whether including  $\text{num}$  in the subset can achieve the current sum  $j$  based on previous computations.

### Final Result:

After completing the dynamic programming steps, the value of  $Q[N][s]$  will be true if and only if there exists a subset in the set that sums to the target value  $s$ .

### Efficiency and Analysis:

The time complexity of this dynamic programming approach is  $O(sN)$ , which is much more efficient than the brute force approach for large target values since it depends on the magnitude of  $s$  rather than the size of the input set  $N$ . This makes it a practical and efficient solution for the subset sum problem in many real-world scenarios. <sup>[6]</sup>

## 3.2 REVIEW OF PAPER #1

*Solving the Subset Sum Problem with Heap-Ordered Subset Trees by Daniel Shea*

### 3.2.1 ABSTRACT

In algorithmic analysis, the subset sum problem is a notable task where, given a set of integers, you determine if any combination can sum to a target value. Various solutions exist beyond brute force, employing clever data structures and approximation methods. This paper introduces a novel approach: a tree-based data structure called the subset tree, built upon the min-heap solution. This structure efficiently solves the subset sum problem for all integers in  $O(N^3 k \log k)$  time, where  $N$  is the set's length and  $k$  is the index of subsets being searched.

### 3.2.2 IMPROVED EXPONENTIAL TIME ALGORITHM

Efforts spanning from 1974 onwards have significantly improved the efficiency of solving the Subset Sum Problem, reducing its complexity from  $O(2^N)$  to  $O(2^{N/2})$ , as demonstrated by Horowitz and Sahni. The algorithm operates by initially partitioning the input set of length  $N$  into two sorted sets, each of size  $N/2$ . Next, it computes and stores sums for all  $2^{N/2}$  possible subsets in separate lists for each set. During initialization, pointers are positioned at the smallest sum in the first list and the largest sum in the second list. The search process involves comparing the sum of these pointers with the target value: a match indicates success, while discrepancies prompt adjustments to the pointers' positions based on whether the sum is less or more than the target. This iterative process continues until the target sum is found or until the search exhausts the available space. These refinements have notably streamlined the Subset Sum Problem's computational demands, making it more tractable for practical applications.

### 3.2.3 ENHANCED $O(2^{N/2})$ SOLUTION WITH MIN-HEAPS AND MAX-HEAPS

Efforts to enhance the Subset Sum Problem's efficiency led to the refinement of the  $O(2^{N/2})$  solution by Schroepel and Shamir in 1981, integrating min-heaps and max-heaps. This enhancement revolutionised subset sum computations, streamlining the process and boosting overall efficiency. The algorithm begins by dividing the sums lists  $A$  and  $B$  into smaller lists, followed by generating and sorting subsets for efficient storage and retrieval using min-heaps and max-heaps. The min-heap prioritises subsets with the smallest total sums, while the max-heap handles subsets with larger total sums. By iteratively adjusting subsets and employing exponential operations, the algorithm efficiently locates subsets that meet the target sum criteria or exhausts search possibilities. The time complexity analysis reveals a significant improvement to  $O(N * 2^{N/2})$ , marking a notable advancement in computational efficiency compared to previous exponential complexities.

### 3.2.4 MIN-HEAP SOLUTION WITH POSITIVE INTEGERS

Use of heap ensures that parent nodes are smaller or equal to their children, forming a partially-ordered structure where the root is the smallest. <sup>[22]</sup>

i. **Algorithm 1** is employed to construct a tree of subsets arranged by increasing sum.

---

**Algorithm 1:** Binary min-heap for subsets of positive integers

---

**Input:**  $S, t$

**Output:**  $O$

- 1 Sort  $S$  into increasing order
  - 2 Define a tree  $O$  where:
    - (I) The root is the singleton consisting only of the first element
    - (II) The left child of a set whose maximum element is  $i$ th in the sorted input list is obtained by replacing that element by element  $i + 1$
    - (III) The right child is obtained by including element  $i + 1$  without removing any existing element
  - 3 Return  $O$
- 

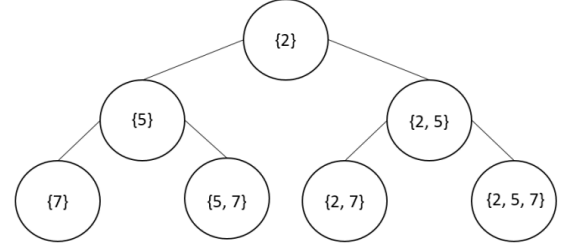


Figure 1: A min-heap binary tree for  $\{2, 5, 7\}$ .

#### ii. Finding the $k$ Smallest Elements in Heap-Ordered Tree:

a) Hierarchical Grouping Method (Fredrickson, 1993):

The algorithm operates with a time complexity of  $O(k)$  by employing hierarchical grouping and a recursively defined data structure within a heap-ordered tree. It efficiently organises specific elements in the heap, allowing for quick retrieval of the  $k$  smallest elements. This design optimises operations involving subsets within larger datasets, making it ideal for scenarios requiring efficient extraction of limited elements. <sup>[10]</sup>

b) Algorithm 2 for Finding  $k$ th Smallest Element:

The algorithm offers an  $O(k \log k)$  time complexity solution for finding the  $k$ th smallest element, presenting an alternative approach to traditional methods. It utilises a lookup strategy within a min-heap binary tree, ensuring efficient retrieval operations with a logarithmic time complexity relative to  $k$ . This method optimises the process of identifying the  $k$ th smallest element, providing a streamlined solution for related computational tasks.

---

**Algorithm 2:** Retrieve the  $k$ th smallest element from a min-heap binary tree

---

**Input:**  $T$

**Output:**  $M[k-1]$

- 1 Define a min-heap of nodes to visit  $V$
  - 2 Insert the root node of  $T$  into  $V$
  - 3 Define an empty array  $M$
  - 4 While  $Length(M) < k$ :
    - (I) Pop the root node of  $V$
    - (II) Add the popped node to  $M$
    - (III) Add the popped node's children to  $V$
-

### iii. Lazy Generation of Child Nodes:

Child nodes are derived from their parent nodes only as needed during the search process. This optimization reduces unnecessary node generation, especially in large min-heap trees

### iv. Binary Search on Sorted Subsets:

- A binary search can be performed on the sorted list of subsets derived from the input set.
- Time Complexity: With  $2^N$  total subsets possible, a binary search to find a target sum  $t$  can be executed in  $O(\log 2^N)$  time. Simplifying to  $O(N \log 2)$ , or  $O(N)$ , for efficient subset sum searches.

### v. Complexity of kth Largest Element Search:

- **Time Complexity:** Each search for the  $k$ th largest element takes  $O(k \log k)$  time.
- **Binary Search on Subsets:** Applying binary search on the virtual sorted list of all subsets for sum retrieval results in  $O(Nk \log k)$  time complexity.

Algorithm 1 becomes inadequate for sets with negative and positive integers as it violates min-heap properties.

### vi. Inclusion of Negative Integers:

Offsetting values by adding the absolute value of the least element plus one ensures all elements become positive, enabling the continued use of a heap-ordered tree for subsets containing positive and negative integers. Storing the offset alongside each element is crucial for maintaining accuracy and efficiency in subset sum computations.

### vii. Challenge with Algorithm 1 for Scaled Sets:

The issue arises when applying Algorithm 1 to scaled sets due to varying target values for subsets of different lengths. For instance, while  $\{-3, -2, 5\}$  sums to 0 in the original set, its scaled version  $\{5, 6, 13\}$  sums to 24 due to the added offset. To resolve this,  $N$  min-heaps are generated to handle scaled subsets consistently, each holding subsets of equal length. Scaling the target sum by  $(\text{offset} * n)$  ensures accurate comparisons during computations, overcoming the inconsistency in target values across subsets.

## 3.2.5 THE HEAP-ORDERED SUBSET TREE

The subset tree is a tree structure that contains subsets of a set  $S$ , with each node representing a subset. It's constructed based on a specific algorithm (Algorithm 3) that creates a heap-ordered tree with  $n$ -length subsets from a set  $S$  of length  $N$ . This algorithm likely outlines steps for building the tree in a way that maintains the heap property, ensuring that nodes are arranged in a specific order based on their values or properties

### i. Total Nodes in Subset Tree:

For a subset tree of order  $n$  created from an input set with  $N$  elements, the total number of nodes in the tree is limited to  $Nn$ . This means that the number of nodes in the tree depends on the size of the input set and the order of the subset tree.



## ii. Maximum Children per Node and Tree Height:

Each node in the subset tree can have at most  $N$  children, representing all possible elements from the input set. Additionally, the height of the tree, which measures how many levels or layers it has, is at most  $N$ . Algorithm 3 constructs a subset tree  $T$  from a set  $S$ , with positive elements, using a scaled set.

---

**Algorithm 3:**  $n$ -subset tree  $T$  from input set  $S$ 

---

**Input:**  $S, n$

**Output:**  $T$

- 1 Sort the input set  $S$
  - 2 Set the root node of the subset tree  $T$  to the  $n$  smallest elements of  $S$
  - 3 The first child node is the subset of the parent node with the parent node's greatest element replaced with the next greatest element in  $S$ . If no such greater element exists, then the end of  $S$  has been reached and this child node should not be generated.
  - 4 Let the variable  $i$  be equal to the penultimate index in the subset of the parent node:  
(I) The next child node is the subset of the parent node with the element at index  $i$  replaced by the next greatest element from index  $i$  in  $S$ . If that element already exists in the subset, then increment the conflicting element with its next greatest element in  $S$ .  
(II) Repeat with any additional conflicts until none exist or there is no greater element in  $S$  with which to increment. If the latter is the case, then the end of  $S$  has been reached and this child node should not be generated.
  - 5 Decrement  $i$  by 1 and repeat Step 4 for all subsequent child nodes until  $i$  is less than the smallest index at which the parent node incremented its subset value
  - 6 Terminate and return  $T$  when the greatest element in the subset is equal to the greatest element in  $S$
- 

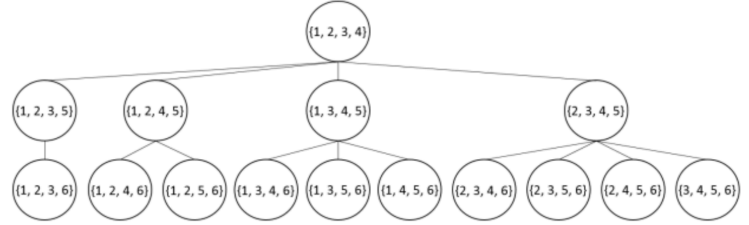


Figure 2: A heap-ordered 4-subset tree for the set  $\{1, 2, 3, 4, 5, 6\}$ .

## iii. Efficient Generation up to the $k$ th Smallest Element:

The process of constructing the subset tree  $T$  from set  $S$  generates the entire tree. However, due to the partial ordering inherent in the construction of the subset tree, it becomes trivial to generate nodes up to the  $k$ th smallest element efficiently. This partial ordering ensures that subsets are arranged in a specific order, making it easier to access elements in sorted order without having to traverse the entire tree.

## iv. Space Optimization for Subset Trees:

By leveraging the partial ordering during subset tree construction, this optimization reduces the space required to store subset trees. Which is particularly advantageous for an arbitrarily large input set.

Working upon the above insights a new approach called Heap-Ordered Subset Trees is derived to solve the subset sum problem. <sup>[1]</sup>

### 3.3 REVIEW OF PAPER #2

*A Fast Heuristic Algorithm for Solving High-Density Subset-Sum Problems by Akash Nag*

#### 3.3.1 ABSTRACT

The subset sum problem is to decide whether for a given set of integers  $A$  and an integer  $S$ , a possible subset of  $A$  exists such that the sum of its elements is equal to  $S$ . The problem of determining whether such a subset exists is NP-complete; which is the basis for cryptosystems of knapsack type. In this paper, a fast heuristic algorithm is proposed for solving subset sum problems in pseudo-polynomial time. Extensive computational evidence suggests that the algorithm almost always finds a solution to the problem when one exists. The runtime performance of the algorithm is also analysed.

#### 3.3.2 INTRODUCTION

The subset sum problem is a NP-complete problem and is defined as follows: Given a set  $A = \{a_i : 1 \leq i \leq n\}$  of integers (usually called weights) and an integer  $S$  (called the sum), determine whether or not there exists a subset  $B$  of  $A$ , such that:

$$\sum_{i=1}^k b_i = S \quad \forall b_i \in B, k = |B|, B \subseteq A$$

For the purposes of this paper, the bounds on the weights is defined to be the range  $R$  of the set, i.e.  $(1 - \frac{R}{2}) \leq a_i \leq (\frac{R}{2} + 1) \forall 1 \leq i \leq n$ . A novel algorithm, called the CP Algorithm (Card-Players' Algorithm), is proposed in the next section for solving the subset-sum problem.

#### 3.3.3 HARDNESS OF THE SUBSET SUM PROBLEM

The hardness of solving the subset-sum problem varies directly with the density of the set of weights. There can be two cases:

- D ≤ 1:** These low-density subset-sum problems are efficiently solved by reduction to a short vector in a lattice, as presented by Brickell; Lagarias and Odlyzko ; Martello and Toth; Coster et al.
- D > 1:** These medium and high-density subset-sum problems are solvable by dynamic programming techniques or using analytic number theory, such as those presented in Chaimovich et al. ; Galil and Margalit ; Flaxman and Przydatek ; with some of these failing to find a solution if certain bounds for  $n$  or  $R$  are not respected.

In this paper, a novel, heuristic but non-deterministic algorithm is proposed for the latter case of high density problems.

Table 1. Average Run-Times (Milliseconds) and Outer-Loop Run Counts\*

$n$	Run-times (milliseconds)			Outer-loop run-counts		
	$R$					
	100	1,000	10,000	100	1,000	10,000
50	0.000	0.078	1.062	6.347	38.393	396.106
100	0.000	0.078	1.326	5.713	21.122	204.929
200	0.015	0.078	1.249	6.010	13.296	96.133
500	0.062	0.172	1.484	8.457	11.286	41.506
1,000	0.140	0.374	2.466	12.250	12.222	27.421
2,000	0.514	0.514	4.632	17.692	15.515	22.653
5,000	2.139	1.621	11.499	32.099	23.741	26.841
10,000	7.971	4.620	21.200	57.264	33.753	34.755

\* Averaged over 1000 instances, each instance consisting of a set of  $n$  random integers in the interval  $[1-(R/2), (R/2)-1]$

### 3.3.4 COMPARISON WITH RELATED RESEARCH ON HIGH DENSITY PROBLEMS

#### 1) Solving Dense Subset-Sum Problems by Using Analytical Number Theory by Mark Chaimovich, Gregory Freiman and Zvi Galil <sup>[19]</sup>

Their paper focussed on high density as a necessary, sometimes sufficient, condition for a discrete problem to admit analytical methods for its solution.

**However, their algorithm had three restrictions:**

1. The first was that the numbers must be distinct. It wasn't able to find the optimal set when repetitions are allowed.
2. The second restriction (the inequality relating  $m$  and  $I$ ) was the density assumption on which the algorithm was based.
3. They hadn't been able to relax the third restriction and handle small and large (close to  $I_m$ ) target numbers  $M$ .

#### 2) An Almost Linear-Time Algorithm for the Subset-Sum Problem by Zvi Galil Oded Margalit <sup>[20]</sup>

The paper introduces a new approach to solving a subset-sum problem, which is beneficial for NP-hard integer programming problems. It provides an algorithm for the dense version of the subset-sum problem, with a time complexity of  $O(l \log l)$  and  $O(m)$  for dense inputs and target numbers near the middle sum. It is always two orders of magnitude faster than the algorithm using dynamic programming. Additionally, it presents a characterization of subset sums as arithmetic progressions, derived using elementary number theory and algorithmic techniques, surpassing previous methods reliant on analytic number theory.

**Problems with this approach:**

The constants in the algorithm and the theorem are quite large. They claimed that the true conditions for our algorithm are much weaker and that large constants are not needed. However, these have not been calculated yet.

#### 3) Solving Medium-Density Subset Sum Problems in Expected Polynomial Time by Abraham D. Flaxman and Bartosz Przydatek <sup>[21]</sup>

This paper was published in 2004. However, all known algorithms at that time for dense SSP took at least  $\Omega(2^m)$  time, and no polynomial time algorithm was known which solved SSP with significantly lower density. They presented an expected polynomial time algorithm for solving uniformly random instances of subset sum problem over the domain  $Z_{2^m}$ , with  $m = O(\log n^2 / \log(\log n))$ . This is possibly the first algorithm working efficiently beyond the magnitude bound of  $O(\log n)$ , thus narrowing the interval with hard-to-solve SSP instances.

**Problems with this approach:**

1. Dense SSP is a deterministic algorithm which can fail with non-zero probability.
2. A natural open question is whether the bound on the magnitude can be further extended, e.g. up to  $(\log n)^z$  for some  $z \geq 2$ . <sup>[2]</sup>

### 3.4 REVIEW OF PAPER #3

*Subset Sum and the Distribution of Information by Daan Van Den Berg and Pieter Adriaans*

#### 3.4.1 ABSTRACT

The subset sum problem's complexity isn't just about lacking an exact subexponential algorithm; it also hinges on the number of bits in the input integers. Empirical data suggests that complexity varies based on how informational bits are distributed among the integers. Hard instances often have equally dispersed bits, while easier ones have more eccentric distributions. Instances with integers matching their bit positions mark a transition in hardness. This indicates a clear link between information dispersal and subset sum problem difficulty, especially in cases with identical  $n$  and  $m$  values.

#### 3.4.2 INTRODUCTION

There are two main versions to solving the subset problem:

1. **Decision Version:** When you can find a subset that adds up to a certain number.
2. **Optimization Version:** When you want to find the subset that comes closest to the target. This version is also known as *Partition Problem*.

##### **Partition Problem**

In this we divide a set of numbers into two groups such that both groups have roughly the same total sum. Consider the set of numbers: {3, 7, 5, 11, 2}. Their total sum numbers is 28. It can be split into 2 groups such that both groups have a sum as close to 14 as possible.

One possible partition could be: {3, 11} and {7, 5, 2}. Both groups have a sum of 14.

#### 3.4.3 METHODS

##### **1. Branch and Bound**

There are two primary forms: queue-based and stack-based. The latter is used in this case.

When the algorithm is approximating  $t$ , and has a current value  $cur$  higher than  $t$ , no additional items are added to it, using  $t$  as a bound when branching through the search space. It traverses all  $2^{|S|}$  sums, but omits branches that cannot possibly result in a better value by saving significant computation time. Finally, the boolean variable `targetFound` ensures the algorithm halts as soon as the first exact solution is found.

##### **2. Templates & Instances**

The algorithm is tested within 70 instances, each having 12 positive integers. These instances are grouped into 7 sets (cohorts), with each set containing 10 instances.

A "strict template" is used to create these instances. This template assigns a specific number of bits to each integer in the instance

Templates range from linearly increasing (like 1b, 2b, 3b...12b) to flat (like 78 bits divided evenly among 12 integers). Templates in the upper half are more eccentric but not extreme.

Table 1: Seven ‘strict templates’ used for making 70 subset sum instances. A value such as  $4b$  randomly generates a corresponding integer of exactly 4 bits, meaning it is randomly chosen between 8 and 16. The templates vary in eccentricity,  $ST_3$  being the most eccentric, and  $ST_{-3}$  being the flattest possible. For each template, one corresponding instance is given as an example.

	Strict Template	Example Instance
$ST_3$	(1b,1b,1b,1b,1b,4b,4b,5b,9b,13b,17b,21b)	{0,1,1,1,0,10,12,17,478,7899,90607,1638220}
$ST_2$	(1b,1b,2b,2b,3b,4b,5b,6b,9b,12b,15b,18b)	{1,1,3,3,6,15,23,40,423,3422,24181,251636}
$ST_1$	(1b,1b,2b,4b,4b,5b,6b,7b,9b,11b,13b,15b)	{0,1,3,8,14,30,45,79,324,1145,4332,19120}
$ST_0$	(1b,2b,3b,4b,5b,6b,7b,8b,9b,10b,11b,12b)	{1,2,6,12,19,35,115,247,305,563,1534,3828}
$ST_{-1}$	(3b,3b,4b,4b,5b,6b,7b,8b,8b,9b,10b,11b)	{7,6,9,13,16,55,109,175,230,330,909,1686}
$ST_{-2}$	(4b,4b,5b,5b,6b,6b,7b,7b,8b,8b,9b,9b)	{11,11,30,26,49,49,84,80,166,156,484,317}
$ST_{-3}$	(6b,6b,6b,6b,6b,6b,7b,7b,7b,7b,7b,7b)	{58,54,35,61,50,49,122,71,111,119,108,92}

### 3.4.4 METHOD OUTCOME:

#### i. Fixed Hardness:

All the eccentric subset sum instances in this experiment are equally difficult, requiring 2048 recursions (repetitive operations).

There can be only one integer  $s_i \geq \frac{1}{2} \sum_{i=1}^{|S|} x_i$  and such an integer is present in all instances generated from templates  $ST_3, ST_2, ST_1$  and  $ST_0$ .

#### ii. Preprocessing Effect:

Sorting the numbers from largest to smallest makes the algorithm skip half of the search tree in the first step itself, making it faster.

#### iii. Key Observation:

In these instances, there's always one big number that's at least half of the total sum. This sets a limit on how many repetitions the algorithm needs to find a solution.

#### iv. Variation in Complexity:

As we move through different templates, some instances start showing variations in complexity. This happens because the biggest number doesn't always make up half of the total sum. <sup>[3]</sup>

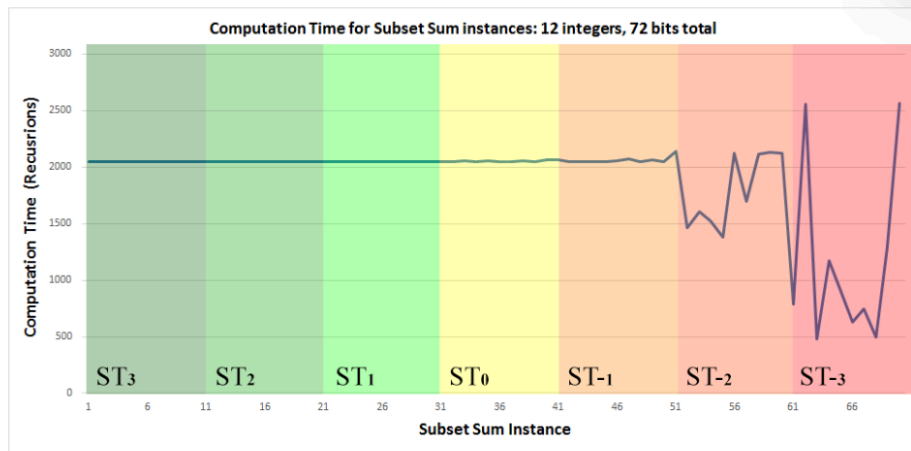


Figure 1: Strict templates of different eccentricity have different hardness patterns. Templates more eccentric than the linear template  $ST_0$  have no variation in computational hardness. Templates less eccentric than  $ST_0$  have both harder and easier instances.

### 3.5 REVIEW OF PAPER #4

*An  $O(\ln n)$  Parallel Algorithm for the Subset Sum Problem by Zhang, Guo Qiang*

#### 3.5.1 ABSTRACT

The Subset Sum Problem (SSP) poses the challenge of determining whether a subset within a given set of numbers can sum up to a specified target sum. Traditionally, solving SSP with sequential algorithms has been difficult due to its inherent complexity. However, the advent of VLSI technology has spurred interest in devising parallel and approximation algorithms for SSP. Notable examples include Karnin's algorithm, which offers a reasonable product of time and processor count but remains impractical in terms of actual time required. Another promising approach by Peters and Rudolph is particularly efficient when the relative error threshold is not near zero. This paper addresses the pressing need for faster parallel algorithms for SSP by proposing a novel parallel algorithm designed to achieve logarithmic parallel time with a high number of processors. Leveraging the Parallel Random Access Machine (P-RAM) model for synchronous parallel computation, this algorithm aims to significantly reduce computation times for SSP, contributing to the advancement of parallel computation and its application in solving complex combinatorial optimization problems like SSP. [12]

#### 3.5.2 INTRODUCTION

The Subset Sum Problem (SSP) is a well-known combinatorial optimization problem with significant practical applications in various fields, including cryptography, task scheduling, and computational biology. Given a finite set  $S = \{1, 2, \dots, n\}$  with associated weights  $a_i \in \mathbb{Z}^+$  for each  $i \in S$ , and a positive integer  $b$ , the SSP involves determining whether there exists a subset  $S' \subseteq S$  such that the sum of the weights of its elements equals  $b$ . This problem is NP-complete, indicating its computational complexity and the difficulty of finding an efficient sequential algorithm to solve it.

#### 3.5.3 PREVIOUS TECHNIQUES AND APPROACHES

Traditionally, sequential algorithms have been used to tackle SSP, but their efficiency diminishes as the size of the problem increases. However, recent advancements in VLSI technology have sparked interest in developing parallel and approximation algorithms for SSP. Karnin proposed a parallel algorithm that achieves a time complexity of  $O(2^{n/2})$  and requires  $O(2^{n/6})$  processors. Similarly, Peters and Rudolph introduced a parallel algorithm that solves SSP with a relative error  $\epsilon$  in polynomial time, demonstrating promising results when sufficient processors are employed. Despite these advancements, existing parallel algorithms still face practical limitations in terms of computational time. [4]

### 3.6 REVIEW OF PAPER #5

*'Ads' Algorithm for Subset Sum Problem by Adarsh Mukar Verma*

#### 3.6.1 ABSTRACT

The Subset Sum Problem is a significant challenge in fields such as complexity theory, bin packing, and cryptography. This problem is known to be NP-complete. In this paper, we present a new technique to solve the Subset Sum Problem. While there have been numerous algorithms proposed based on greedy strategies, lattice-based reductions, and other methods, the approach we introduce here is grounded in straightforward mathematical concepts and utilises binary search.

The current upper bound for Subset Sum is apparently  $O(2^{N/2})$  when size of the input set (denoted  $n$ ) is used as the complexity parameter. When the maximum value in the set (denoted  $m$ ) is used as the complexity parameter, dynamic programming can be used to solve the problem in  $O(m^3)$  time. There are several ways to solve SSP in exponential and polynomial time. A naive algorithm with time complexity  $O(2^N)$  solves SSP by iterating all the possible subsets and each for its subset comparing its sum with target  $X$ . A better algorithm proposed a scheme which achieves time  $O(N2^{N/2})$ . If the target  $T$  is relatively small then there exist dynamic programming algorithms that can run much faster.

In this paper we'd study about a simple SSP solution based on arithmetic and binary search with  $O(N^2 \log n)$  time complexity and linear space complexity.

#### 3.6.2 PRE-REQUISITE KNOWLEDGE

##### 1. Sorting

For the subset sum problem we need to arrange the elements in non-decreasing order, then subset sum problem can be solved by the 'Ads' algorithm. Sorting of the list can be done by various algorithms of sorting and the complexity depends on the type of algorithm we are using. If there are less elements in the list then we can use a simple sorting algorithm like Bubble sort with worst case complexity  $O(N^2)$  or we can use an efficient quicksort with average case complexity  $O(n \log n)$  or merge sort with complexity  $O(n \log n)$  for more numbers.

##### 2. Binary Search

Binary Search or half-interval search algorithm search the specified value within the sorted array. It is based on a divide and conquer approach. Binary search will require far fewer comparisons than linear search, if the list to be searched contains more than a dozen elements. But it imposes the requirement that the list should be sorted. Binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time. The worst case performance of binary search is  $O(\log n)$  and best case is  $O(1)$ , And the worst space complexity is  $O(1)$ . [5]

## 4. IMPLEMENTATION AND RESULTS

### 4.1 IMPLEMENTATION ENVIRONMENT

#### For Java:

**Programming Language:** The algorithm was implemented using Java, specifically Java Standard Edition 8 with Update 5.

**Java Version:** This implementation utilised Java SE 8 (Update 5), which provides features and improvements over previous versions.

**Processor:** The code execution occurred on an Intel Pentium Dual-Core processor.

**Processor Speed:** The processor's clock speed for executing the Java code was 2.3GHz.

#### For C++:

**Programming Language:** The algorithm was coded in C++, adhering to the C++14 standard for language features and compatibility.

**Compiler Standard:** The C++ code was compiled using a compiler that supports the C++14 standard, ensuring proper execution and adherence to language specifications.

**Processor:** The execution of the C++ code took place on an Intel Pentium Dual-Core processor.

**Processor Speed:** The processor operated at a clock speed of 2.3GHz during the execution of the C++ algorithm.



## 4.2 INDIVIDUAL PAPER RESULTS

### 4.2.1 PAPER #1

*Solving the Subset Sum Problem with Heap-Ordered Subset Trees by Daniel Shea*

#### ALGORITHM:

**Input:**  $S, t$   
**Output:**  $R$

- 1 Sort  $S$
- 2 Offset each element in  $S$  by the absolute value of the least element in  $S$  plus one
- 3 Store this *offset* value
- 4 Define a variable  $O$  and set it to 1. This is the order of the virtual subset heap being searched.
- 5 While  $O \leq N$  and a subset  $R$  that sums to  $S + (\text{offset} * O)$  has not been found:
  - (I) Perform a binary search for  $S + (\text{offset} * O)$  on the  $O$ -subset tree  $T$
  - (II) If a subset  $R$  is found, then terminate the loop. Else, increment  $O$  by 1 and repeat Step 5(I). Do this until a subset has been found or  $O > N$
- 6 If a subset  $R$  has been found, then subtract *offset* from each element in  $R$  and return  $R$ . Else, return an empty subset.

In this algorithm, we first sort the numbers in set  $S$  in ascending order and then make all numbers positive by adding the absolute value of the smallest number plus one to each number. We initialise a variable  $O$  to 1, representing the size of subsets we're looking for. We search for a subset sum by incrementing  $O$  until we find a subset that sums up to  $S + (\text{offset} * O)$ , where the offset is the value added to make numbers positive. This search utilises a binary search within the subset tree of order  $O$ . If a subset with the target sum is found, we convert it back to its original values by subtracting the offset, yielding the result  $R$ . If no such subset is found after searching all possible subset sizes, we return an empty subset.

#### Algorithm Example

Returning to the previous example of the input set  $\{-7, -3, -2, 5, 8\}$  and the target value of 0, the algorithm is executed as follows:

1.  $\{-7, -3, -2, 5, 8\}$  is sorted. In this case, the set was already sorted.
2. An offset of 8 is applied to  $\{-7, -3, -2, 5, 8\}$ , producing a scaled set of  $\{1, 5, 6, 13, 16\}$ .
3.  $O$  is set to 1.
4. A binary search for  $S + (\text{offset} * O)$ , or 8, is performed on the subset tree of order  $O$ , or 1.
5. Since no subset is found,  $O$  is incremented by 1. As  $O \not\geq 5$ , the search is continued.
6. A binary search for  $S + (\text{offset} * O)$ , or 16, is performed on the subset tree of order  $O$ , or 2.
7. Since no subset is found,  $O$  is incremented by 1. As  $O \not\geq 5$ , the search is continued.
8. A binary search for  $S + (\text{offset} * O)$ , or 24, is performed on the subset tree of order  $O$ , or 3.
9. The subset  $\{5, 6, 13\}$  is found. Subtract *offset* from each element to produce the subset  $\{-3, -2, 5\}$ .
10. Return  $\{-3, -2, 5\}$ .

## CODE:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.PriorityQueue;

public class SubsetSum {
    public ArrayList<Integer> input = new ArrayList<Integer>();
    public ArrayList<Integer> scaledInput = new ArrayList<Integer>();
    public TreeNode<List<Integer>> tree;
    /* This code comes from https://github.com/vivin/GenericTree
    It is used as the generic node for a subset tree
    */
    public SubsetSum() {
        input.add( -7 );
        input.add( -3 );
        input.add( -2 );
        input.add( 0 );
        input.add( 1 );
        input.add( 5 );
        input.add( 8 );
        input.add( 17 );
        input.add( 21 );

        // The sum for which we want to find an appropriate subset
        int target = 50;

        // 1. Sort the set
        Collections.sort( input );

        // Scale the set so that there are no negative elements, with the offset
        // being stored in a variable
        int offset = 0;
        if ( input.get( 0 ) <= 0 ) {
            offset = Math.abs( input.get( 0 ) ) + 1;
            for ( int i = 0; i < input.size(); ++i ) {
                scaledInput.add( input.get( i ) + offset );
            }
        }

        // Search all n-sized min-heaps from set S for the target sum
        int scaledTarget;
        boolean targetFound = false;
        List<Integer> result = new ArrayList<Integer>( );
        int resultSum = 0;
        for ( int n = 1; n <= input.size(); ++n ) {
            // Scale the target to be searchable in the n-sized Subset Tree
            scaledTarget = target + ( offset * n );
```

```

    // Now generate the k'th smallest subset
    tree = buildTree( scaledInput, n );
    result = binarySearch( tree, 0, choose( input.size( ), n ), scaledTarget );
    if ( result == null ) {
        continue;
    }
    if ( sum( result ) == scaledTarget ) {
        targetFound = true;
        break;
    }
}

// Output the result
if ( targetFound ) {
    for ( int i = 0; i < result.size( ); ++i ) {
        result.set( i, result.get( i ) - offset );
    }
    System.out.println( "Subset that sums to the target sum has been found!" );
    printList( result );
}
else {
    System.out.println( "No subset sums to the target sum " + target );
}
}

public List<Integer> binarySearch( TreeNode<List<Integer>> tree, int lowerBound,
int upperBound, int target ) {
    if ( lowerBound > upperBound ) {
        return null;
    }
    int position = Math.round( ( lowerBound + upperBound ) / 2.0f );
    if ( position == 0 ) {
        return null;
    }
    List<Integer> kthMin = findKthMin( tree, position );
    int sum = sum( kthMin );
    if ( lowerBound == upperBound ) {
        if ( sum == target ) {
            return kthMin;
        }
        else {
            return null;
        }
    }
    else {
        if ( sum == target ) {
            return kthMin;
        }
    }
}

```

```

    }
    else if ( sum < target ) {
        if ( lowerBound == position ) {
            position += 1;
        }
        return binarySearch( tree, position, upperBound, target );
    }
    else {
        if ( upperBound == position ) {
            position -= 1;
        }
        return binarySearch( tree, lowerBound, position, target );
    }
}
}

public TreeNode<List<Integer>> buildTree( ArrayList<Integer> list, int n ) {
    TreeNode<List<Integer>> tree = new TreeNode<List<Integer>>();

    ArrayList<Integer> idcs = new ArrayList<Integer>();
    for ( int i = 0; i < n; ++i ) {
        idcs.add( i );
    }

    // 2. Set the root node to the n smallest elements of S
    tree.setData( list.subList( 0, n ) );
    tree.setIndices( idcs );
    tree.setLimit( 0 );

    return tree;
}

/*
 * In this case, the algorithm terminates after the first layer of children are
 * created;
 * the recursive case is not run
 */
public List<TreeNode<List<Integer>>> buildChildren( List<Integer> list,
List<Integer> indices, int limit ) {
    ArrayList<TreeNode<List<Integer>>> children = new
ArrayList<TreeNode<List<Integer>>>();

    for ( int i = list.size( ) - 1; i >= limit; --i ) {
        if ( indices.get( indices.size( ) - 1 ) == input.size( ) - 1 ) {
            continue;
        }
        TreeNode<List<Integer>> child = new TreeNode<List<Integer>>();

```

```

        ArrayList<ArrayList<Integer>> tmpArr = incrementList( list, indices, i );
        child.setData( tmpArr.get( 0 ) );
        child.setIndices( tmpArr.get( 1 ) );
        child.setLimit( i );
        children.add( child );
    }

    return children;
}

public ArrayList<ArrayList<Integer>> incrementList( List<Integer> list,
List<Integer> indices, int idx ) {
    if ( list.size( ) < 1 ) {
        return null;
    }
    ArrayList<Integer> newList = deepCopy( list );
    ArrayList<Integer> newIndices = deepCopy( indices );
    if ( list.size( ) == 1 ) {
        if ( idx < list.size( ) ) {
            newList.set( idx, scaledInput.get( indices.get( idx ) + 1 ) );
            newIndices.set( idx, indices.get( idx ) + 1 );
        }
        else {
            return null;
        }
    }
    else {
        do {
            newList.set( idx, scaledInput.get( indices.get( idx ) + 1 ) );
            newIndices.set( idx, indices.get( idx ) + 1 );
            ++idx;
        } while ( idx < list.size( ) && newIndices.get( idx - 1 ) == indices.get(
idx ) );
        if ( newIndices.get( newIndices.size( ) - 1 ) == newIndices.get(
newIndices.size( ) - 2 ) ) {
            return null;
        }
    }
    ArrayList<ArrayList<Integer>> retVal = new ArrayList<ArrayList<Integer>>( );
    retVal.add( newList );
    retVal.add( newIndices );
    return retVal;
}

public ArrayList<Integer> deepCopy( List<Integer> list ) {
    ArrayList<Integer> newList = new ArrayList<Integer>( );
    for ( int i = 0; i < list.size(); ++i ) {
        newList.add( list.get( i ) );
    }
}

```

```

    }
    return newList;
}

@SuppressWarnings("unchecked")
public List<Integer> findKthMin( TreeNode<List<Integer>> tree, int k ) {
    Comparator<TreeNode<List<Integer>>> comparator = new NodeComparator();
    PriorityQueue<TreeNode<List<Integer>>> toVisit = new
PriorityQueue<TreeNode<List<Integer>>>( 11, comparator );
    TreeNode<List<Integer>> root = new TreeNode<List<Integer>>(
tree.getData( ) );
    root.setIndices( tree.getIndices( ) );
    root.setLimit( tree.getLimit( ) );
    root.setChildren( tree.getChildren( ) );
    toVisit.add( root );
    ArrayList<TreeNode<List<Integer>>> smallestNodes = new
ArrayList<TreeNode<List<Integer>>>( );
    while ( smallestNodes.size( ) < k ) {
        TreeNode<List<Integer>> node = toVisit.poll( );
        List<TreeNode<List<Integer>>> children = buildChildren( node.getData( ),
node.getIndices( ),
                                                                    node.getLimit( ) );
        for ( TreeNode<List<Integer>> child : children ) {
            TreeNode<List<Integer>> newChild = new TreeNode<List<Integer>>(
child.getData( ) );
            newChild.setIndices( child.getIndices( ) );
            newChild.setLimit( child.getLimit( ) );
            newChild.setChildren( child.getChildren( ) );
            toVisit.add( newChild );
        }
        smallestNodes.add( node );
    }
    return smallestNodes.get( k - 1 ).getData( );
}

public int sum( List<Integer> node ) {
    int sum = 0;
    for ( Integer n : node ) {
        sum += n;
    }
    return sum;
}

public void printList( List<Integer> list ) {
    for ( int i = 0; i < list.size( ); ++i ) {
        System.out.print( list.get( i ) + "\t" );
    }
    System.out.println( );
}

```

```

}

public static int choose( int n, int k ) {
    if (k > n) {
        return 0;
    }
    if (k == 0) {
        return 1;
    }
    if (k > n / 2) {
        return choose(n, n - k);
    }
    return n * choose(n - 1, k - 1) / k;
}

public static void main(String[] args) {
    new SubsetSum( );
}
}

```

## OUTPUT:

```

Subset that sums to the target sum has been found!
-2      1      5      8      17     21

```

### **TIME COMPLEXITY:**

The algorithm's time complexity is  $O(N^3 k \log k)$  where , N is the set length (number of integers) and k is the index of the subset tree during binary search.

**Sorting Complexity:** Sorting set S of length N takes  $O(N \log N)$  time using efficient algorithms like quicksort or mergesort.

**Binary Search Complexity:** Binary search on subsets of size n (order of the subset tree) has a time complexity of  $O(n \log N)$  due to binomial coefficients.

**Heapify and Lookup Complexity:** Lookup operations take  $O(N \log k)$  time per operation. The overall complexity for each lookup becomes  $O(N^2 k \log k)$ .

**Worst-case Iteration Complexity:** When no subset exists for the target sum, running the  $O(N^2 k \log k)$  operation N times leads to an overall complexity of  $O(N^3 k \log k)$  for the algorithm.

### **SPACE COMPLEXITY:**

Considering the above components, the overall space complexity is the sum of these major contributors:

**$O(n)$  (input data) +  $O(n)$  (scaled input) +  $O(m)$  (tree nodes) +  $O(\text{temporary space})$  +  $O(k)$  (priority queue).**

The exact space complexity can vary based on the input size, the structure of the subset tree, and the execution flow during runtime. Ignoring constant factors and lower-order terms, we can represent the overall space complexity as  **$O(\max(n, m, k, \text{temporary space}))$ .**



## 4.2.2 PAPER #2

### *A Fast Heuristic Algorithm for Solving High-Density Subset-Sum Problems by Akash Nag*

#### ALGORITHM:

The proposed algorithm is a non-deterministic iterative algorithm, and is hereafter called the Card-Players' Algorithm (CP Algorithm), after the manner in which the solution is obtained. The algorithm is best explained using an analogy of two card players: Alice and Bob, playing a game of cards. The deck of cards with which the game is played is not the usual 52-card deck, but is rather an  $n$ -card deck with each card numbered with an integer  $a$  in the range  $[1-(R/2), (R/2)-1]$ . The input to the algorithm, which is the set of integers  $A$ , is therefore analogous to the deck of cards in this game. The game is a challenge game, where Bob picks a number  $S$  (or the sum, being the other input to the algorithm) at the beginning of the game, and challenges Alice to come up with a hand in which the cards add up to  $S$ .

#### 1. Initialization

The game begins by shuffling the deck, and then dealing it equally among Alice and Bob, whereupon each end up with  $n/2$  cards in hand. If  $n$  is odd, Alice will end up having one more card than Bob. Bob then chooses a random number  $S(obj)$ , and challenges Alice to come up with a hand in which the cards add up to  $SOBJ$ . Formally, let  $H_{Alice}$  and  $H_{Bob}$  be the two hands of Alice and Bob,  $A$  be the total deck of cards, and  $n$  be the total number of cards in the deck. At the beginning of the game, these are defined as:

$$\begin{aligned} A &= \{a_i : 1 \leq i \leq n\} & \forall a_i &\in [1 - R/2, R/2 - 1] \\ H_{Alice} &= \left\{ x_i : 1 \leq i \leq \left\lfloor \frac{n}{2} \right\rfloor + n \bmod 2 \right\} & \forall x_i &\in A \\ H_{Bob} &= \left\{ y_i : 1 \leq i \leq \left\lfloor \frac{n}{2} \right\rfloor \right\} & \forall y_i &\in A, H_{Alice} \cap H_{Bob} = \phi \end{aligned}$$

#### 2. Objective

The objective of the game, for Alice, is to win the challenge thrown by Bob, i.e. to come up with a sum of  $SOBJ$ . During the entire duration of the game, Bob only plays to help Alice achieve her goal, and has no separate objective of his own.

#### 3. Game Play

Prior to the beginning of each turn, an ordered pair formed from the sum of Alice's hand and that of Bob's hand are recorded. Whenever, after a turn, such a sum-pair is repeated in the game, a fair coin is tossed, and depending on the outcome – either Alice or Bob randomly chooses a card from her/his hand and gives it to the other. After such a transfer, the sum-pair is again checked for repetition and if it repeats, the coin toss is repeated again until either the number of such consecutive tosses equals  $n$ , or a unique sum-pair is obtained. The game proceeds if a unique sum-pair is obtained; otherwise, Alice loses the game and declares that she cannot win the challenge. Let  $SAlice$  and  $SBob$  define the sum of all the cards in Alice's and Bob's hand respectively. Then the sum-pair ( $SAlice$ ,  $SBob$ ) must be unique at every turn of the game. These are defined as follows:

$$\begin{aligned} S_{Alice} &= \sum_{i=1}^{n(H_{Alice})} x_i & \forall x_i &\in H_{Alice} \\ S_{Bob} &= \sum_{i=1}^{n(H_{Bob})} y_i & \forall y_i &\in H_{Bob} \end{aligned}$$

At each turn, Alice makes her decision based on two values: her requirement (denoted by REQ), and the Closest-Element (denoted by CE). The former is calculated by Alice herself, while she asks Bob for the latter. These are defined as:

$$\begin{aligned}
 REQ &= S_{OBJ} - S_{Alice} \\
 CE &= \begin{cases} y_k & f(REQ) \neq \phi, d_k = \min(d_i) \forall (y_i, d_i) \in f(REQ), y_i \in H_{Bob} \\ \phi & f(REQ) = \phi \end{cases} \\
 f(p) &= \{(y, d) : SIGN(y, p) = 1, |y| \leq |p|, d = \|y\| - |p|\} \\
 SIGN(x, y) &= \begin{cases} 1 & x < 0, y < 0 \\ 1 & x \geq 0, y \geq 0 \\ 0 & otherwise \end{cases}
 \end{aligned}$$

Informally, Alice computes what card she needs to achieve her objective, and then asks Bob if he has that card (i.e. CE). Bob then checks his hand to see if he has that card. If yes (i.e.  $CE \neq \phi$ ), he gives that card to Alice. If he does not have such a card (i.e.  $CE = \phi$ ), Alice checks how many cards she has in her hand. If she has only 1 card left, she asks Bob for a random card from his hand. If she has more than 1 card left, she calculates the Appropriate Discardable Element (ADE) as follows:

$$\begin{aligned}
 G &= \{(s_i, x_i) : s_i = S_{Alice} - x_i \forall x_i \in H_{Alice}\} \\
 ADE &= \begin{cases} x_k & s_k = \min(s_i) \forall (s_i, x_i) \in G \\ \phi & G = \phi \end{cases}
 \end{aligned}$$

If no such element is present (i.e.  $ADE = \phi$ ), Alice loses the game, and declares that she cannot win the challenge. If an ADE does exist however, she gives that card to Bob instead, thus completing the current turn. At the end of each turn, the winning condition is checked – if  $S_{Alice} = S_{OBJ}$ , then Alice wins the challenge and the solution to the subset-sum problem is Alice's hand  $H_{Alice}$ . If not, the game continues. The algorithm is presented below.

```

ALGORITHM CP(a[0..n-1], target)
begin
    sumList=new HashSet();
    t=(n/2)+(n mod 2);
    x[ ]=a[0..t-1];
    y[ ]=a[t..n-1];
    sx=SUM(x), sy=SUM(y);

    while (sx ≠ target) do
        begin
            rc=0;
            while (SEARCH(sumList, sx, sy) ≠ -1) do
                begin
                    rc=rc+1;
                    randomly transfer 1 element from x to y or vice-versa and update sx, sy accordingly
                    if (rc=n) then return FAILURE;
                end
                k=GetRequirement(x, target); // see (3.3)
                cei=GetClosestElementIndex(y, k); // see (3.3)
                if (cei>=0) then
                    transfer y[cei] to x and update sx, sy
                else
                    if (LENGTH(x)>1) then
                        di=GetAppropriateDiscardIndex(x); // see (3.4)
                        if (di=-1) then return FAILURE;
                        transfer x[di] to y and update sx, sy
                    else
                        randomly transfer 1 element from y to x and update sx, sy
                    end if
                end if
                add (sx, sy) to sumList;
            end
            return x;
        end

```

---

**CODE:**

```
import java.util.*;

class Pair
{
    protected int x;
    protected int y;

    public Pair(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    @Override
    public int hashCode()
    {
        int hash = 7;
        hash = 97 * hash + this.x;
        hash = 97 * hash + this.y;
        return hash;
    }

    @Override
    public boolean equals(Object obj)
    {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;

        final Pair other = (Pair) obj;
        if (this.x != other.x) return false;
        if (this.y != other.y) return false;

        return true;
    }
}

public class CardPlayersAlgorithm
{
    private static int[] generate(int n, int min, int max)
    {
        int a[] = new int[n];
        for(int i=0; i<n; i++)
        {
            a[i] = min + (int)((max-min+1) * Math.random());
        }
        return a;
    }
}
```

```

private static boolean search(HashSet<Pair> sumList, int sx, int sy)
{
    Pair sp = new Pair(sx, sy);
    return sumList.contains(sp);
}

private static int sign(int x, int y)
{
    if(x<0 && y<0)
        return 1;
    else if(x>=0 && y>=0)
        return 1;
    else
        return 0;
}

private static HashSet<Pair> getRequirement(int p)
{
    HashSet<Pair> set = new HashSet<Pair>();
    int abp = Math.abs(p);
    for(int y=-abp; y<=abp; y++)
    {
        if(sign(y,p)==1)
        {
            int d = Math.abs(Math.abs(y) - abp);
            set.add(new Pair(y,d));
        }
    }
    return set;
}

private static int searchInHand(ArrayList<Integer> hand, int elem)
{
    int n = hand.size();
    for(int i=0; i<n; i++)
    {
        if(elem==hand.get(i)) return i;
    }
    return -1;
}

private static int getClosestElementIndex(ArrayList<Integer> hBob,
HashSet<Pair> req)
{
    if(req.size()==0) return -1;
    Iterator<Pair> it = req.iterator();
    Pair minPair = null;
    int yIndex = -1;

```

```

while(it.hasNext())
{
    Pair pair = it.next();
    int search = searchInHand(hBob,pair.x);

    if(search > -1)
    {
        if(minPair == null)
        {
            minPair = pair;
            yIndex = search;
        } else {
            if(pair.y <= minPair.y)
            {
                minPair = pair;
                yIndex = search;
            }
        }
    }
}

return yIndex;
}

private static int getMaxInHand(ArrayList<Integer> hand)
{
    int n = hand.size();
    int s = hand.get(0);
    for(int i=1; i<n; i++)
    {
        if(s >= hand.get(i)) s = hand.get(i);
    }
    return s;
}

private static int getAppropriateDiscardIndex(ArrayList<Integer> hAlice, int
sx)
{
    int max = getMaxInHand(hAlice);
    HashSet<Pair> set = new HashSet<Pair>();
    int n = hAlice.size();

    Pair minPair = null;
    int minIndex = -1;

    for(int i=0; i<n; i++)
    {
        int e = hAlice.get(i);
        int s = sx - e;
    }
}

```

```

        Pair p = new Pair(s,e);

        if(minPair == null) {
            minPair = p;
            minIndex = i;
        } else if(p.x <= minPair.x) {
            minPair = p;
            minIndex = i;
        }
    }

    if(max!=hAlice.get(minIndex))
    {
        System.out.print("FALSE: ["+max+", " + sx+"]: ");
        for(int i=0; i<hAlice.size(); i++)
            System.out.print(hAlice.get(i)+",");
        System.out.println();
    }
    return minIndex;
}

private static ArrayList<Integer> getSubset(int a[], int n, int target)
{
    int t = (n/2) + (n % 2);
    int sx = 0, sy = 0;

    HashSet<Pair> sumList = new HashSet<Pair>();
    ArrayList<Integer> x = new ArrayList<Integer>(t);
    ArrayList<Integer> y = new ArrayList<Integer>(t);

    for(int i=0; i<t; i++)
    {
        sx += a[i];
        x.add(a[i]);
    }

    for(int i=t; i<n; i++)
    {
        sy += a[i];
        y.add(a[i]);
    }

    while(sx != target)
    {
        int rc = 0;
        while(search(sumList, sx, sy))
        {
            rc++;
            int direction = (int)(100 * Math.random());

```

```

        if(direction >= 50)
        {
            int len = y.size();
            int choice = (int)(len * Math.random());
            int element = y.get(choice);
            y.remove(choice);
            x.add(element);
            sy -= element;
            sx += element;
        } else {
            int len = x.size();
            int choice = (int)(len * Math.random());
            int element = x.get(choice);
            x.remove(choice);
            y.add(element);
            sx -= element;
            sy += element;
        }

        if(rc==n) return null;
    }

    HashSet<Pair> k = getRequirement(target - sx);
    int cei = getClosestElementIndex(y, k);
    if(cei >= 0)
    {
        int element = y.get(cei);
        y.remove(cei);
        x.add(element);
        sy -= element;
        sx += element;
    } else {
        if(x.size() > 1)
        {
            int di = getAppropriateDiscardIndex(x, sx);
            if(di == -1) return null;

            int element = x.get(di);
            y.add(element);
            x.remove(di);
            sy += element;
            sx -= element;
        } else {
            int len = y.size();
            int choice = (int)(len * Math.random());
            int element = y.get(choice);
            y.remove(choice);
            x.add(element);
            sy -= element;

```

```

        sx += element;
    }
}

sumList.add(new Pair(sx,sy));
}
return x;
}

public static void main(String args[])
{
    int n=6;
    int target=13;
    int a[] = {0,1,5,8,17,21};
    for(int e:a) System.out.print(e+", ");
    System.out.println("\nTarget="+target);
    ArrayList<Integer> result = getSubset(a,n,target);
    if(result==null) System.out.println("Failure");
    int len = result.size();
    if(len==0) System.out.println("Failure");
    System.out.println("Subset with the given target sum is: ");
    for(int i=0; i<len; i++) System.out.print(result.get(i)+"", " ");
    System.out.println();
}
}

```

## OUTPUT:

```

0, 1, 5, 8, 17, 21,
Target=13
FALSE: [0,6]: 0,1,5,
FALSE: [0,22]: 0,1,21,
FALSE: [0,13]: 8,0,5,
FALSE: [0,25]: 0,8,17,
FALSE: [0,29]: 0,8,21,
FALSE: [0,29]: 0,8,21,
FALSE: [0,9]: 0,8,1,
FALSE: [0,26]: 8,1,0,17,
FALSE: [0,26]: 8,1,0,17,
FALSE: [0,30]: 8,1,0,21,
FALSE: [0,14]: 8,1,0,5,
FALSE: [0,27]: 1,0,5,21,
FALSE: [1,14]: 1,5,8,
FALSE: [1,23]: 1,5,17,
FALSE: [8,25]: 8,17,
Subset with the given target sum is:
8, 5, 0,

```



### **TIME COMPLEXITY:**

The author believes that much faster run-times for this algorithm are possible if the algorithm is implemented in C/C++ and additional optimizations are applied.

The outer-loop run-count is indeterminate and is dependent on both  $n$  and  $R$ , while the combined inner processes have time complexity  $O(n)$ ,

Hence the time complexity of the CP Algorithm is  **$O(c.n)$**  for some  $c$  dependent on  $n$  and  $R$ .

The large values for  $n$  and the non-deterministic nature of the algorithm, made it impossible to perform an exhaustive search (using the power-set) to check indeed that no solution exists when the algorithm reported failure. But considering the fact that only very seldom were failures reported, it is strongly believed that the algorithm almost always succeeds.

### **SPACE COMPLEXITY:**

**HashSet<Pair> sumList:** This HashSet stores pairs of integers representing subset sums.

**ArrayList<Integer> x and y:** These ArrayLists store elements from the input array  $a$  and are used to generate subset sums. In the worst case, each ArrayList could contain all elements of the input array  $a$ , which would result in a space complexity of  $O(n)$  for each ArrayList.

Considering both the HashSet and the ArrayLists, the dominant factor in space complexity is the HashSet `sumList` because it potentially grows quadratically with the input size. Therefore, the overall space complexity of the given code can be approximated as  $O(N^2)$ .

### 4.2.3 PAPER #3

*Subset Sum and the Distribution of Information by Daan Van Den Berg and Pieter Adriaans*

#### ALGORITHM:

1. Sort  $S$  in descending order, assign  $cur=0$ ,  $i=1$ ,  $i_{max}=|S|$ ,  $best=\infty$ ,  $t=\frac{1}{2}\sum_{i=1}^{|S|}x_i$ ,  $targetFound = false$
2. Assign  $cur_1 = cur + s_1$ ,  $cur_2 = cur$ .  
If  $cur_1 = t$ , then assign  $targetFound=true$   
Else  $best = cur_1$ .
3. If  $i < i_{max}$  and  $cur_1 < t$  and  $\neg targetFound$  : go to 2 with  $cur = cur_2$ ,  $i=i+1$
4. If  $i < i_{max}$  and  $cur_2 < t$  and  $\neg targetFound$  : go to 2 with  $cur = cur_2$ ,  $i=i+1$ .

This algorithm, called branch and bound, is used to solve problems like the subset sum problem efficiently. Imagine you have a set of numbers and you want to find a subset that adds up to a specific target number.

**Basic Idea:** The algorithm starts by sorting the numbers in descending order and sets up some variables like  $cur$  (current sum),  $i$  (current index),  $best$  (best sum found so far),  $t$  (target sum), and  $targetFound$  (a flag to stop when a solution is found).

**Main Loop:** It then goes through a loop where it adds each number to the current sum ' $cur$ ' one by one and checks if  $cur$  reaches the target  $t$ . If it does, it sets  $targetFound$  to true and stops.

**Pruning:** The clever part is how it prunes branches in the search tree. It only continues adding numbers if it's possible to reach the target  $t$ . If adding the current number would make  $cur$  higher than  $t$ , it doesn't add it because it's not going to lead to a better solution.

**Stopping Criteria:** The algorithm stops as soon as it finds a subset that adds up to the target  $t$ , ensuring it's solution-optimal for optimization problems.

## CODE:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric> // Include numeric header for accumulate function
#include <climits>

using namespace std;
bool targetFound = false;
vector<int> solution;

void branchAndBoundSubsetSum(const vector<int>& S, int cur, int i, int imax, int&
best, int t, vector<int>& currentSet) {
    if (i >= imax || cur > t || targetFound) {
        return;
    }
    int cur1 = cur + S[i];
    int cur2 = cur;

    if (cur1 == t) {
        targetFound = true;
        solution = currentSet;
        solution.push_back(S[i]); // Include current element in the set
        return;
    }

    if (cur1 <= t && cur1 < best) { // Adjusted condition to prioritize subsets
closer to the target
        best = cur1;
    }

    if (i + 1 < imax && cur1 <= t && !targetFound) { // Adjusted condition for
exploring subsets closer to the target
        currentSet.push_back(S[i]); // Include current element in the set
        branchAndBoundSubsetSum(S, cur1, i + 1, imax, best, t, currentSet);
        currentSet.pop_back(); // Backtrack and remove current element from the
set
    }
    if (i + 1 < imax && cur2 <= t && !targetFound) { // Adjusted condition for
exploring subsets closer to the target
        branchAndBoundSubsetSum(S, cur2, i + 1, imax, best, t, currentSet);
    }
}

int main() {
    vector<int> S = {2, 3, 7, 8, 10};
    sort(S.rbegin(), S.rend()); // Sort in descending order

    int cur = 0, i = 0, imax = S.size(), best = INT_MAX;
    int t = accumulate(S.begin(), S.end(), 0) / 2; // Target value
```

```

vector<int> currentSet;

branchAndBoundSubsetSum(S, cur, i, imax, best, t, currentSet);

if (targetFound) {
    cout << "Exact solution found! Subset with sum equal to target:" << endl;
    for (int num : solution) {
        cout << num << " ";
    }
    cout << endl;
} else {
    cout << "No exact solution found." << endl;
}
return 0;
}

```

## OUTPUT:

```

Exact solution found! Subset with sum equal to target:
10 3 2

```

## TIME COMPLEXITY:

The time complexity of this algorithm depends on the number of subsets generated and the operations performed in each recursive call.

In the worst case, the algorithm explores all possible subsets of the input set.

Since each element can either be included or excluded in a subset (binary choice), there are  $2^n$ , possible subsets for a set of size  $n$ .

Therefore, the time complexity of the algorithm is  $O(2^n)$ , where  $n$  is the size of the input set.

## SPACE COMPLEXITY:

The space complexity of the algorithm depends on the space required for the recursive function calls, the auxiliary data structures used (like vectors), and the depth of the recursion.

In each recursive call, additional space is used for parameters, local variables, and the vector storing the current subset.

The depth of recursion can be at most equal to the size of the input set, i.e.,  $n$ .

Therefore, the space complexity of the algorithm is  $O(n)$  in terms of auxiliary space used during the recursion.

## 4.2.4 PAPER #4

*An  $O(\ln n)$  Parallel Algorithm for the Subset Sum Problem by Zhang, Guo Qiang*

### ALGORITHM:

The Shift Triangle Algorithm: The Shift Triangle Algorithm presents a novel approach to solving SSP using parallel computation. This algorithm leverages the concept of shift operations on sets of integers, known as the Shift Operation  $*$ . Given an integer sequence  $a_i$  with  $1 < i < 2^k$ , the algorithm constructs a Shift Triangle, which represents a sequence of integer sets obtained through successive shift operations. The Shift Triangle culminates in  $A_{k+1}$  the last term of the triangle, which holds crucial information for solving SSP efficiently. [13]

1. **Input:** Finite set  $S = \{1, 2, 3, \dots, n\}$ , weights  $a_i$  for  $1 < i < n$ , target sum  $b$
2. **Output:** True if there exists a subset  $S' \subseteq S$  such that  $\sum_{i \in S'} a_i = b$ , False otherwise

#### Step 0: Initialization

- Calculate  $t = \sum_{i=1}^n a_i - 2b$

#### Step 1: Shift Operations

- For  $i$  from 1 to  $2k - 1$ ,  $l$  from 1 to 2,  $m$  from 1 to  $\frac{2n}{2}$ 
  - If  $l = 1$ , set  $B[i, 1, m] = A[2i - 1] + A[2i]$
  - If  $l = 2$ , set  $B[i, 1, m] = A[2i - 1] - A[2i]$

#### Step 2: Update Local Memory

- For  $i$  from 1 to  $2k - 1$ ,  $l$  from 1 to 2,  $m$  from 1 to  $\frac{2n}{2}$ 
  - If  $l = 1$ , set  $A[2i - 1] = B[i, 1, m]$
  - If  $l = 2$ , set  $A[2i] = B[i, 1, 1]$

#### Step 3: Further Shift Operations

- For  $i$  from 1 to  $2k - 2$ ,  $l$  from 1 to 4,  $m$  from 1 to 2
  - If  $1 < l < 2$ , set  $B[i, 1, m] = A[4(i - 1) + 1] + A[4(i - 1) + m + 2]$
  - If  $3 < l < 4$ , set  $B[i, 1, m] = A[4(i - 1) + 1 - 2] - A[4(i - 1) + m + 2]$

#### Step 4: Update Local Memory

- For  $i$  from 1 to  $2k - 2$ ,  $l$  from 1 to 4,  $m$  from 1 to 2
  - If  $1 < l < 2$ , set  $A[8(i - 1) + 2(1 - 1) + m] = B[i, 1, m]$
  - If  $3 < l < 4$ , set  $A[8(i - 1) + 4 + 2(1 - 3) + m] = B[i, 1, m]$

#### Step 5: Continue Shift Operations

- For  $t$  from 1 to  $2k$ 
  - For  $i$  from 1 to  $2k - t$ ,  $l$  from 1 to  $2^{2t-1}$ ,  $m$  from 1 to  $2^{2t-1} - 1$ 
    - If  $1 < l < 2^{2t-1} - 1$ , set  $A[2^{2t-1}(i - 1) + 2^{2t-1} - 1(l - 1) + m] = B[i, 1, m]$
    - If  $2^{2t-1} - 1 + 1 < l < 2^{2t-1}$ , set  $A[2^{2t-1}(i - 1) + 2^{2t-2} + 2^{2t-1}(l - 2^{2t-1} - 1) + m] = B[i, 1, m]$

#### Step 6: Final Decision

- For  $i$  from 1 to  $2k$ ,  $l$  from 1 to  $2^{2k-1}$ ,  $m$  from 1 to  $2^{2k-1} - 1$ 
  - If  $1 < l < 2^{2k-1} - 1$  and  $|B[i, 1, m]| = |t|$ , set  $L = true$

[14]

## CODE:

```
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

// Function to calculate the Shift Triangle
vector<int> calculateShiftTriangle(const vector<int>& A) {
    int n = A.size();
    vector<int> B(n);

    // Calculate Shift Triangle
    for (int i = 0; i < n - 1; ++i) {
        for (int l = 0; l < 2; ++l) {
            for (int m = 0; m < n / 2; ++m) {
                if (l == 0) {
                    B[i] += A[2 * i] + A[2 * i + 1];
                } else if (l == 1) {
                    B[i] += A[2 * i] - A[2 * i + 1];
                }
            }
        }
    }
    return B;
}

// Function to check if subset with sum equal to 'target' exists using Shift
Triangle Algorithm
bool subsetSum(const vector<int>& weights, int target) {
    int n = weights.size();
    int k = log2(n) + 1;

    // Calculate t
    int t = 0;
    for (int i = 0; i < n; ++i)
        t += weights[i];
    t -= 2 * target;

    // Initialize A
    vector<int> A = weights;

    // Calculate Shift Triangle
    for (int t = 1; t <= k; ++t) {
        for (int i = 0; i < n - t; ++i) {
            A = calculateShiftTriangle(A);
        }
    }
}
```

```

    // Check if |t| exists in A
    for (int i = 0; i < n; ++i) {
        if (abs(A[i]) == abs(t)) {
            return true;
        }
    }

    return false;
}

int main() {
    vector<int> weights = {3, 1, 4, 2};
    int b = 5;

    if (subsetSum(weights, b))
        cout << "Subset sum exists for b = " << b << endl;
    else
        cout << "Subset sum doesn't exist for b = " << b << endl;

    return 0;
}

```

#### OUTPUT:

```
Subset sum exists for b = 5
```

#### TIME COMPLEXITY:

Initialization:  $O(n)$

Shift Operations:  $O(k^2)$

Overall time complexity:  $O(k^2)$ , where  $k = \log_2 n$

#### SPACE COMPLEXITY:

Global Memory (A):  $O(n)$

Global Memory (L):  $O(1)$

Local Memory (B):  $O(n)$

Overall space complexity:  $O(n)$

## 4.2.5 PAPER #5

### *“ADS” algorithm for the Subset Sum Problem by Adarsh Kumar Verma*

#### **ALGORITHM:**

With the Subset Sum problem, however, we do not find a mutual dependence between the number of objects in the set and the Target X. So, the procedure is, first we have to sort the given list to solve it by Ads algorithm or we can say the proposed algorithm is only for sorted array or list. Suppose if the list is not already sorted then we can sort it by efficient sorting algorithms. After sorting we will apply the Ads algorithm for solving the subset sum problem. The algorithm uses simple mathematics, like, if sum of two numbers A, B is C then we can find B by subtracting it from C, such that  $B = C - A$ . Similarly if the sum of three numbers A, B, C from the set  $\{ A, B, C, D \}$  is X then we can also find the C by subtracting  $(A + B)$  from X such that  $C = X - (A + B)$ . And we can search C by binary search from the sorted list. We will apply binary search in the list from the element 3 of the list, because we have already taken A, B. Searching will be in the remaining set  $\{ C, D \}$ . Hence the set will be  $\{ A, B, C \}$  whose sum is X.

#### **3.1 Pseudo code for ‘Ads’ algorithm**

Input : Set of n sorted positive integers  $e = \{ e_1, e_2, e_3, \dots, e_n \}$   
where  $e_1, e_2, e_3, e_4, \dots, e_n$  are in non-decreasing order.

Target positive integer X.

ads ( e, X )

```
1.  for ( i = 0 ; i < n-1 ; i++)
2.      sum = 0
3.      for ( j = i ; j < n - 1 ; j++)
4.          sum = sum + e [ j ]
5.          c = X - sum
6.          if ( c > e [ j ] )
7.              flag = binarySearch ( c, j, n-1 )
8.              if flag = 1
9.                  “ subset found with target sum X”
10.                 for ( k = i, l=0 ; k <= j ; k++, l++)
11.                     s [ l ] = e [ k ]
12.                     l++
13.                     s [ l ] = c
14.                     display( s )
15.                 else
16.                     break
17.  if flag = - 1
18.  “ no subset found”
```

#### **3.2 Pseudo code for binary search :**

Input: array e , lower bound, upper bound

binarySearch ( c, j, n-1)

```
1.  low = j,      high = n - 1
2.  while ( low <= high )
3.      mid = ( low + high ) / 2
4.      if ( c >= e[ mid ] )
5.          if ( c == e[ mid ] )
6.              return ( 1 )
7.          else
8.              low = mid + 1
9.
10.         else
11.             high = mid - 1
12.  return( - 1 )
```



**EXAMPLE:**

Problem : - Given a set  $e = \{ 2, 3, 6, 7, 10 \}$  and Target sum  $X = 13$ .

Find a subset whose sum is equal to  $X$ .

Solution :- Number of elements in set  $n = 5$ , we are considering the array from 0 to 4.

Pass 1 :  $i = 0$

sum=0 ,

1.  $j = i = 0, e[0] = 2$

sum = sum +  $e[j] = 0 + 2 = 2$

$c = X - \text{sum} = 13 - 2 = 11$

check if (  $11 > 2$  ) , yes

flag = binarySearch(c)

binarySearch(11) from index 1 to 4

flag = -1

2.  $j = 1, e[1] = 3$

sum = sum +  $e[j] = 2 + 3 = 5$

$c = X - \text{sum} = 13 - 5 = 8$

check if (  $8 > 3$  ), yes

flag = binarySearch(c)

binarySearch(8) from index 2 to 4

flag = - 1

3.  $j = 2, e[2] = 6$

sum = sum +  $e[j] = 5 + 6 = 11$

$c = X - \text{sum} = 13 - 11 = 2$

check if (  $2 > 3$  ), no

Break.

Pass 2 :

$i = 1$

sum = 0

1.  $j = i = 1, e[1] = 3$

sum = sum +  $e[j] = 0 + 3 = 3$

$c = X - \text{sum} = 13 - 3 = 10$

check if (  $10 > 3$  ), yes

flag = binarySearch(c)

binarySearch(10) from index 2 to 4

flag = 1

hence, subset found  $S = \{ 3, 10 \}$

2.  $j = 2, e[2] = 6$

sum = sum +  $e[j] = 3 + 6 = 9$

$c = X - \text{sum} = 13 - 9 = 4$

check if (  $4 > 6$  ), no

break

Pass 3 :

$i = 2$

sum = 0

1.  $j = i = 2, e[2] = 6$

sum = sum +  $e[j] = 0 + 6 = 6$

$c = X - \text{sum} = 13 - 6 = 7$

check if (  $7 > 6$  ), yes

flag = binarySearch(c)

binarySearch(7) from index 3 to 4

flag = 1

hence, subset found  $S = \{ 6, 7 \}$

2.  $j = 3, e[3] = 7$

sum = sum +  $e[j] = 6 + 7 = 13$

$c = X - \text{sum} = 13 - 13 = 0$

check if (  $0 > 6$  ), no

break ,

Pass 4 :

$i = 3$

sum = 0

1.  $j = i = 3, e[3] = 7$

sum = sum +  $e[j] = 0 + 7 = 7$

$c = X - \text{sum} = 13 - 7 = 6$

check if (  $6 > 7$  ), no

break.

Hence by the suggested algorithm we get two subsets whose sum is  $X = 13$  and subsets are  $\{ 3, 10 \}$  and  $\{ 6, 7 \}$ .

## CODE:

```
#include <iostream>
#include <vector>
#include <algorithm>

// Function to perform binary search
int binarySearch(const std::vector<int>& e, int c, int j, int n) {
    int low = j;
    int high = n;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (c >= e[mid]) {
            if (c == e[mid]) {
                return 1; // Element found
            }
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1; // Element not found
}

// Function to find a subset with target sum X
void findSubset(const std::vector<int>& e, int X) {
    int n = e.size();
    std::vector<int> s; // Array to store the subset

    // Outer loop to iterate through the array
    for (int i = 0; i < n - 1; i++) {
        int sum = 0;

        // Inner loop to calculate the sum
        for (int j = i; j < n - 1; j++) {
            sum += e[j];
            int c = X - sum;

            // Check if c is greater than the current element
            if (c > e[j]) {
                int flag = binarySearch(e, c, j + 1, n - 1);

                if (flag == 1) {
                    std::cout << "Subset found with target sum X\n";
                    // Store the subset elements
                    for (int k = i, l = 0; k <= j; k++, l++) {
                        s.push_back(e[k]);
                    }
                }
            }
        }
    }
}
```

```

        s.push_back(c);
        // Display the subset
        std::cout << "Subset: ";
        for (int num : s) {
            std::cout << num << " ";
        }
        std::cout << std::endl;
        return;
    }
} else {
    break; // Break inner loop if c is not greater than e[j]
}
}

// If no subset is found, print a message
std::cout << "No subset found with target sum X" << std::endl;
}

int main() {
    // Example array and target sum
    std::vector<int> e = { 2, 3, 6, 7, 10};
    int X = 117;

    // Find subset with target sum X
    findSubset(e, X);

    return 0;
}

```

## OUTPUT:

```

Subset found with target sum X
Subset: 7 10

```

### **TIME COMPLEXITY:**

In computing time complexity, one good approach to count primitive operations. Some examples of primitive operations are assigning value to a variable, indexing into an array, calling a method, performing an arithmetic operation, returning from a method.

The time complexity of 'Ads' algorithm depends on the two for loops and the method binary search, first for loop runs 0 to  $(n - 1)$  in  $i$ , similarly second for loop runs  $j = i$  to  $(n - 1)$ . Hence due to two for loops, primitive operations will occur  $(n - 1) \times (n - 1)$ , that will be  $O(n^2)$ . But due to binary search inside the for loops it will be  **$O(n^2 \log(n))$** .

### **SPACE COMPLEXITY:**

Space complexity analysis was critical in the early days of computing when storage space on the computers was limited. When considering space complexity, algorithms are divided into those that need extra space to do their work.

The space complexity of Ads algorithm is linear  **$O(n)$**  due to the single array required for its execution.

## 5. COMPARATIVE RESULTS

### 5.1 ON THE BASIS OF TIME AND SPACE COMPLEXITIES:

S.No.	Algorithm	Time Complexity	Space Complexity
1.	Heap-Ordered Subset Trees	$O(N^3 k \log k)$	$O(n)$
2.	Card Player Algorithm	$O(c.n)$	$O(n^2)$
3.	Branch and Bound Algorithm	$O(2^n)$	$O(n)$
4.	Shift Triangle Algorithm	$O(k^2)$ , where $k = \log_2 n$	$O(n)$
5.	'Ads' Algorithm	$O(N^2 \log n)$	$O(n)$

## 5.2 ON THE BASIS OF THE OUTPUT PROVIDED ON THE SAME TEST CASE:

```
Test case array: {0,1,5,8,17,21}
Target Sum: 13
```

### ALGORITHM 1

```
Subset that sums to the target sum has been found!
5      8
```

### ALGORITHM 2

```
Target=13
FALSE: [0,6]: 0,1,5,
FALSE: [0,29]: 0,8,21,
FALSE: [0,9]: 0,8,1,
FALSE: [0,29]: 0,8,21,
FALSE: [0,25]: 8,0,17,
FALSE: [0,25]: 8,0,17,
FALSE: [0,9]: 8,0,1,
FALSE: [0,22]: 0,1,21,
FALSE: [0,18]: 0,1,17,
FALSE: [0,22]: 0,1,21,
FALSE: [0,13]: 0,8,5,
FALSE: [0,22]: 0,5,17,
FALSE: [0,22]: 0,5,17,
FALSE: [0,26]: 0,5,21,
FALSE: [0,25]: 0,8,17,
FALSE: [0,13]: 0,8,5,
FALSE: [0,26]: 0,5,21,
FALSE: [0,6]: 0,5,1,
FALSE: [8,29]: 8,21,
FALSE: [1,30]: 8,1,21,
```

```
FALSE: [1,30]: 8,1,21,
FALSE: [1,14]: 8,1,5,
FALSE: [0,13]: 0,8,5,
FALSE: [8,29]: 8,21,
FALSE: [8,25]: 8,17,
FALSE: [1,26]: 8,1,17,
FALSE: [1,14]: 8,1,5,
FALSE: [1,27]: 1,5,21,
FALSE: [1,23]: 1,5,17,
FALSE: [1,14]: 1,5,8,
FALSE: [1,22]: 1,21,
FALSE: [1,18]: 1,17,
FALSE: [0,30]: 1,8,0,21,
FALSE: [0,13]: 8,0,5,
FALSE: [0,26]: 0,5,21,
FALSE: [0,13]: 0,5,8,
FALSE: [0,22]: 0,5,17,
8, 5, 0,
```

### ALGORITHM 3

```
Exact solution found! Subset with sum equal to target:
8 5
```

### ALGORITHM 4

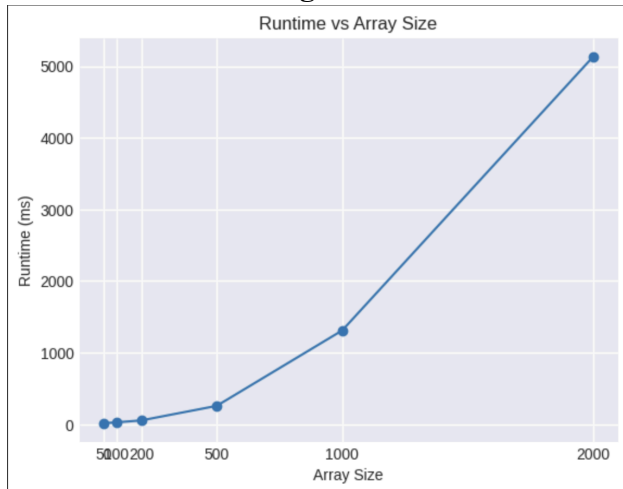
```
/tmp/z7aL2ILRkc.o
Subset sum exists for b=13
```

### ALGORITHM

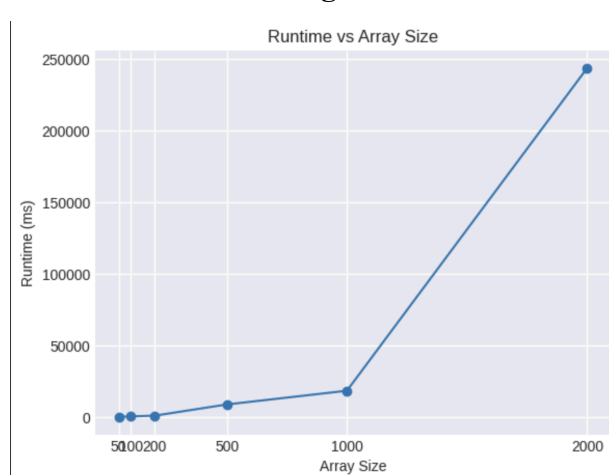
```
Subset found with target sum X
Subset: 5 8
```

## 5.3 ON THE BASIS OF RUNTIME IN COMPARISON WITH SET SIZE

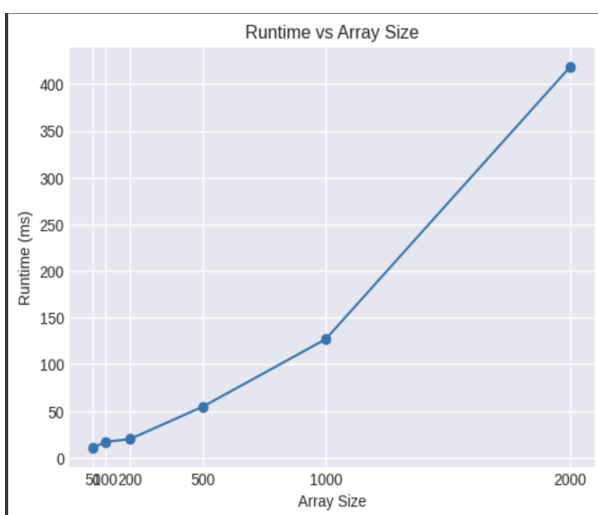
**Algorithm 1**



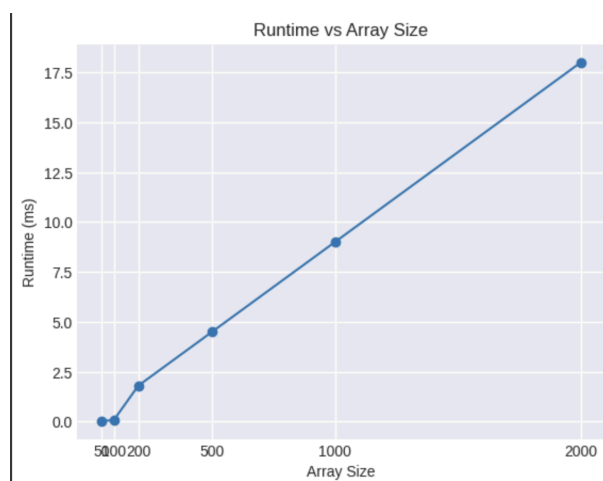
**Algorithm 3**



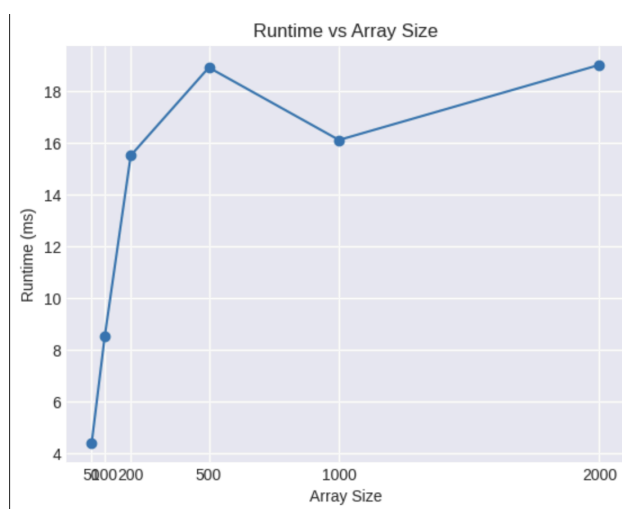
**Algorithm 2**



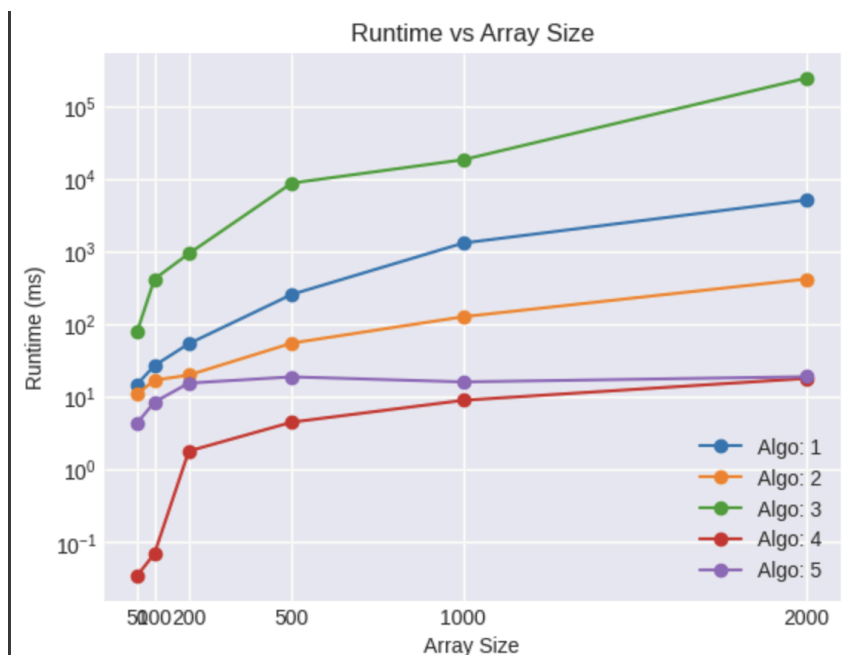
**Algorithm 4**



**Algorithm 5**



## 5.4 PERFORMANCE COMPARISON





## 6. CONCLUSION AND FUTURE

The subset sum problem has undergone a significant evolution, transitioning from brute-force methods to sophisticated dynamic programming and data structure-based approaches.

As evidenced by the review of various papers, researchers have explored diverse strategies to tackle the subset sum problem, addressing its computational complexity and practical applications.

In the first review paper, the subset sum problem has evolved from brute-force methods to dynamic programming and data structure-based approaches, such as min-heaps, for efficient subset searching. This paper introduces the subset tree data structure, which enables proper heap-ordering of subsets regardless of element types. By leveraging subset trees,  $k$ th-minimum lookup operations become feasible, with a time complexity of  $O(N^3 k \cdot \log k)$ , where  $N$  is the set length and  $k$  is the subset index. Future work aims to handle mixed element sets, optimise space usage, refine search strategies, and generalise subset tree algorithms for diverse real-world applications, improving algorithmic performance and scalability.

In the second review paper, the subset-sum problem has garnered significant attention in the computing community. While knapsack-based cryptosystems have suffered setbacks since the compromise of the Merkle-Hellman system, high-density subset sums remain relevant for applications such as computer passwords, message verification, RFID security, and spam filtering. The proposed algorithm provides a promising solution for tackling these challenges.

In the third review paper, the study examines how information distribution impacts the difficulty of subset sum problems. Utilising a branch and bound algorithm, decisions are made incrementally, enabling early search space reduction for instances with highly skewed distributions. Certain patterns of information distribution can significantly influence problem-solving complexity. The paper suggests further research to compare findings with other NP-complete problems and emphasises the importance of understanding information distribution for predicting problem difficulty.

The Shift Triangle Algorithm proposes a promising solution to the Subset Sum Problem (SSP) through parallel computation techniques. With a time complexity of  $O(k^2)$  and space complexity of  $O(n)$ , where  $k = (\log 2n)$ , it outperforms existing methods. However, optimization opportunities remain. Future research could focus on enhancing efficiency, scalability through parallelization, adaptation for various inputs, real-world applications assessment, and hardware implementations. Overall, the Shift Triangle Algorithm shows potential for advancing parallel computation in solving complex optimization problems like SSP.

In the fifth review paper, it can be argued that the Subset Sum problem is easier than the other NP-complete problems, based on algorithms that solve the problem in sub-exponential time. We identify a generic construction of cryptosystems based on the Subset Sum Problem. The suggested approach can be used for the implementation of SSP-based cryptosystems and application of these cryptosystems in defining an efficient RFID (RadioFrequency Identification) security and privacy solutions.

As we move forward, it's evident that the subset sum problem remains a fertile ground for exploration, with implications reaching beyond theoretical computer science into practical domains.

Through ongoing research and collaboration, we can continue to refine algorithms, develop innovative approaches, and unlock new insights into the subset sum problem and its broader implications in computational theory and real-world applications.

## REFERENCES

1. <https://arxiv.org/pdf/1512.01727.pdf>
2. <https://www.mecs-press.net/ijmsc/ijmsc-v3-n2/IJMSC-V3-N2-5.pdf>
3. <https://drive.google.com/file/d/1xfhJQQNQhFIraOM8ACin0VYPEAcz4zTN/view>
4. [https://drive.google.com/file/d/1mPrip2t9rjJFGWpxBRByeKrAg8\\_boobg/view](https://drive.google.com/file/d/1mPrip2t9rjJFGWpxBRByeKrAg8_boobg/view)
5. <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=cd0122f704374d18de8389538>
6. [https://www.researchgate.net/publication/341840033\\_Dynamic\\_Programming\\_for\\_the\\_Subset\\_Sum\\_Problem](https://www.researchgate.net/publication/341840033_Dynamic_Programming_for_the_Subset_Sum_Problem)
7. Koiliaris, Konstantinos; Xu, Chao (2015-07-08). "A Faster Pseudopolynomial Time Algorithm for Subset Sum". arXiv:1507.02318.
8. Michael R. Garey, and David S. Johnson. *Computers and Intractability: A Guide to the theory of NPCompleteness*. WH Freeman & Co., New York. pp:223. 1979.
9. Harsh Bhasin and NehaSingla, "Harnessing Cellular Automata and Genetic Algorithms to solve Travelling Salesman Problem".
10. <https://tinyurl.com/3b4erjmr>
11. [https://en.wikipedia.org/wiki/Subset\\_sum\\_problem](https://en.wikipedia.org/wiki/Subset_sum_problem)
12. M .R. Gray and D .S . Johnson," Computers and Intractability— A Guide to the Theory of NP-Completeness" W .H . FREEDMAN AND COMPANY, 1979
13. E.D . Karnin, A Parallel Algorithm for the Knapsack Problem ,Technical Report, IBM Research Laboratory, California, 1983
14. J. Peters and L . Rudolph, Parallel Approximation Schemes for Subset Sum and Knapsack Problems , Technical Report, Department of Computer Science, Carnegie-Mellon University, 1984
15. <https://www.britannica.com/science/NP-complete-problem>
16. <https://en.wikipedia.org/wiki/NP-completeness>
17. [https://www.researchgate.net/publication/342335335\\_Computational\\_Complexity\\_TheoryPNPNP-Complete\\_and\\_NP-Hard\\_Problems](https://www.researchgate.net/publication/342335335_Computational_Complexity_TheoryPNPNP-Complete_and_NP-Hard_Problems)
18. <https://www.iitg.ac.in/deepkesh/CS301/assignment-2/subsetsum.pdf>
19. [pdf.sciencedirectassets.com](https://pdf.sciencedirectassets.com)
20. AnAlmostLinearTimeAlgo
21. <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=3487ef79fe06c712eae9f14e3bf1278918121874>
22. <https://www.geeksforgeeks.org/introduction-to-min-heap-data-structure/>

