

Data Structures and Algorithms

MOD002641

Element 011

School of Computing and Information Technology

BEng Computer Science

Academic Year: 2023

Trimester: 2

Contents

[Task A – Subset Sum Problem 3](#_Toc132599571)

[A. a) Subset Sum Decision 3](#_Toc132599572)

[A. b) Subset Sum Optimisation 5](#_Toc132599573)

[A. c) Subset Sum Further Optimisation 7](#_Toc132599574)

[Task B – Filial-heir Chains 12](#_Toc132599575)

[1. Formula 1 – Number of Edges 12](#_Toc132599576)

[2. Formula 2 – Number of total nodes 13](#_Toc132599577)

[3. Formula 3 – Number of leaf nodes 14](#_Toc132599578)

[4. Formula 4 – Number of internal nodes 15](#_Toc132599579)

[5. Formula 5 – Min and Max height 16](#_Toc132599580)

[Appendix 17](#_Toc132599581)

[References 22](#_Toc132599582)

Table of Figures

Figure 1 - Task A a) - False Example 3

Figure 2 - Task A a) - False Example – Result 3

Figure 3 - Task A a) - True Example 4

Figure 4 - Task A a) - True Example – Result 4

Figure 5 - Task A b) - Example for smallest available difference 5

Figure 6 - Task A b) - Example for smallest available difference – Result 6

Figure 7 - Task A b) - Example for value 6

Figure 8 - Task A b) - Example for value - Result 7

Figure 9 - Task A c) – Alternative Optimisations - Main 8

Figure 10 - Task A c) – Alternative Optimisations - Main 9

Figure 11 - Task A c) – Alternative Optimisations - Main 10

Figure 12 - Task A c) – Alternative Optimisations - Functions 10

Figure 13 - Task A c) - Alternative Optimisations - Results 11

# Task A – Subset Sum Problem

## A. a) Subset Sum Decision

For the first part of task A, I have chosen to use Java as the programming language. The function is called “isSubsetSum” which takes a subset marked as “array”, an element index number marked “n”, and the sum marked as “sum”. This function iterates through the array over all possible combinations of the array elements that would add up to the sum and return “TRUE” or “FALSE” corresponding to the sum being found equalled or not.

Firstly, the base cases are checked, such as if the sum searched is zero or if the array hold zero elements. Then it’s checked if the last element of the array is larger than the sum, if yes, the last elements are being ignored and the function is called again to check the next element. Finally, check if the sum can be obtained from one of the two OR arguments, including the last checked element or excluding the last checked element.

In Figure 1 and Figure 2 we have the first example of a FALSE result, in this example we have nine array elements and the sum we are looking for is eleven:

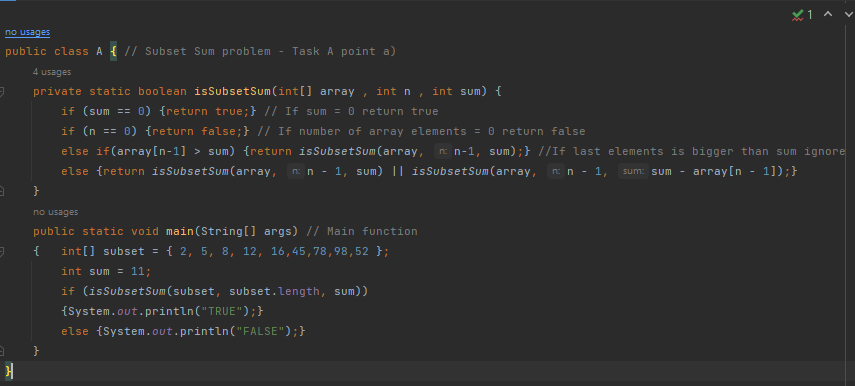


Figure - Task A a) - False Example

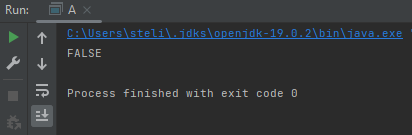


Figure - Task A a) - False Example – Result

For the second example we have a TRUE result found in Figure 3 and Figure 4; the array length and elements are the same and the sum we are searching for is ten:

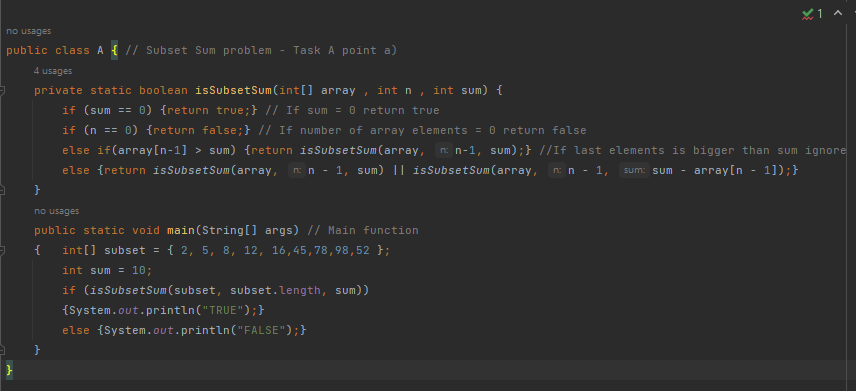


Figure - Task A a) - True Example

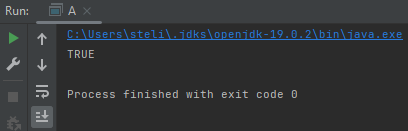


Figure - Task A a) - True Example – Result

## A. b) Subset Sum Optimisation

In the second part of task A, we had to elevate the results from Boolean to integers as we are now looking for the value of the sum or the smallest available difference. Such as if our array A = {1, 2, 5} and our sum t = 4, the smaller available difference equals to 1 + 2 = 3.

For this optimization timings have been included as seen in Figure 5 and Figure 6, which will become more useful on the following part of this task but will show how fast the computation power is with such a small array.

Small changes have been made to the function to accommodate the Boolean change, as well as the last part has now changed from the OR operation to a comparison where the maximum available value smaller or equal to the sum is returned.

Inside the main has been added a display of the sum and the length of the array as well as the results. The results are marked as an integer smaller or equal than the sum and the amount of time taken to process the result. Timer is calculated and converted to seconds, then displayed next to the integer result. This algorithm has an output time complexity of O(), where “n” is the size of the sub sum array.

Text

Description automatically generated

Figure - Task A b) - Example for smallest available difference

In this instance we do not have the correct number for sum = 11, therefore the smallest available difference is found to be 10, and due to the small array size, the optimised function run in less than a second:

A screenshot of a computer

Description automatically generated with medium confidence

Figure - Task A b) - Example for smallest available difference – Result

In Figure 7 we can see the change in the sum from sum = 11 to sum = 21.

Text

Description automatically generated

Figure - Task A b) - Example for value

In this case the sum could be found in the array, and the time did not change as the array size remained the same as seen in Figure 8.

Text

Description automatically generated

Figure - Task A b) - Example for value - Result

## A. c) Subset Sum Further Optimisation

In this part of the task the aim was to describe, develop and test one or more optimisation alternatives that run better than exponential time and can work better with larger arrays that still return the lowest possible difference from the requested sum. Most of the incorporated code is calculating the processing time and displaying the results as seen in the Figure 9, Figure 10 and Figure 11.

This time we have five arrays containing 5, 10, 20, 30 and 3000 integers each. The optimisation discussed at part b) has been added, as seen in Figure 12, but could not be used up to 3000 integers as the time it would take to complete was too large and has not been included in the figures.

The second optimisation named “dynamic” is a copy of the first algorithm, but using 2D arrays, making the length of the array equal to . This algorithms time complexity is O(), where “n” is the size of the sub sum array and “k” is the sum of the elements of the array.

On the last algorithm marked “memorised” where HashMap is used. This option checks the same possibilities of the sum or array being equal to zero, then checks if the controls are used. There are two integer values being kept according to the if cases, one resulting from “exclude” resulting into a reduced index, and the other from “include” where HashMap keeps the key and value. The key is and the value is the “exclude” and “include” values, finally returning the HashMap value.

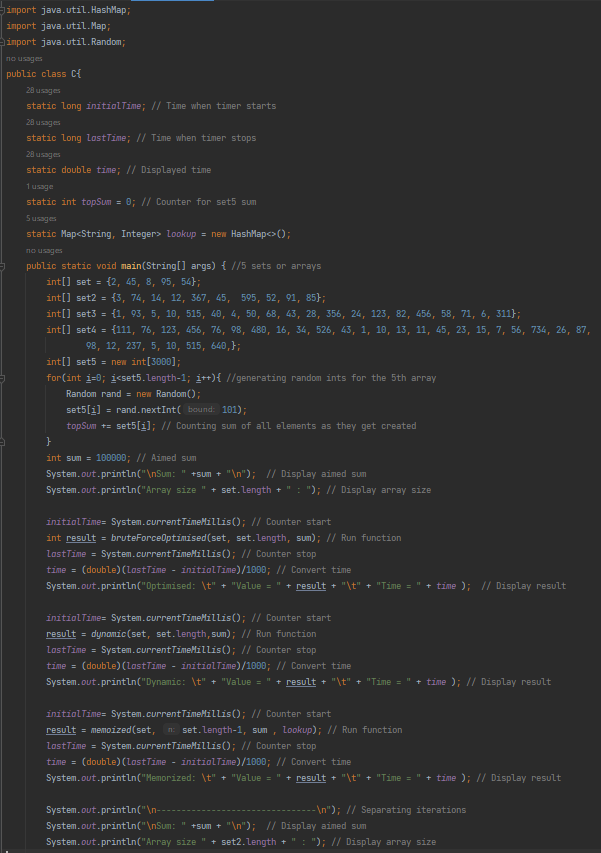


Figure - Task A c) – Alternative Optimisations - Main

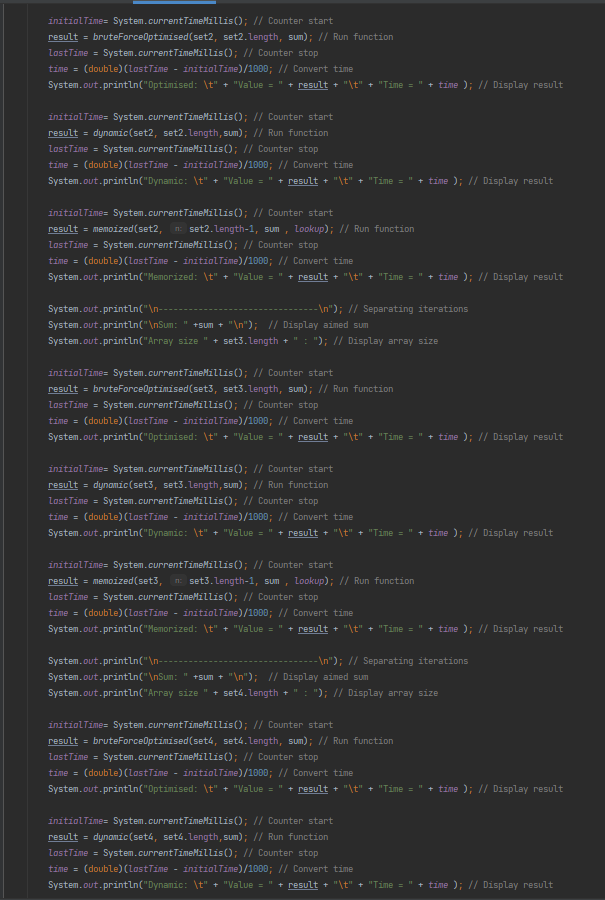


Figure - Task A c) – Alternative Optimisations - Main

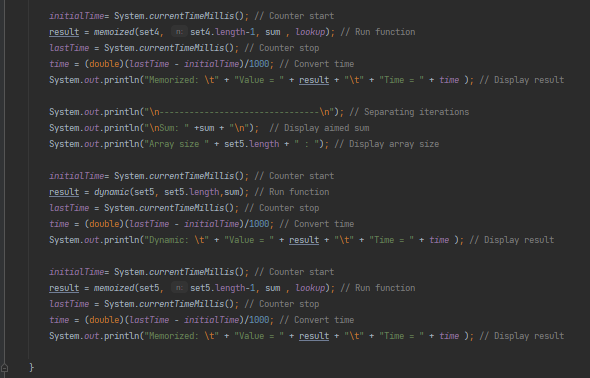


Figure - Task A c) – Alternative Optimisations - Main

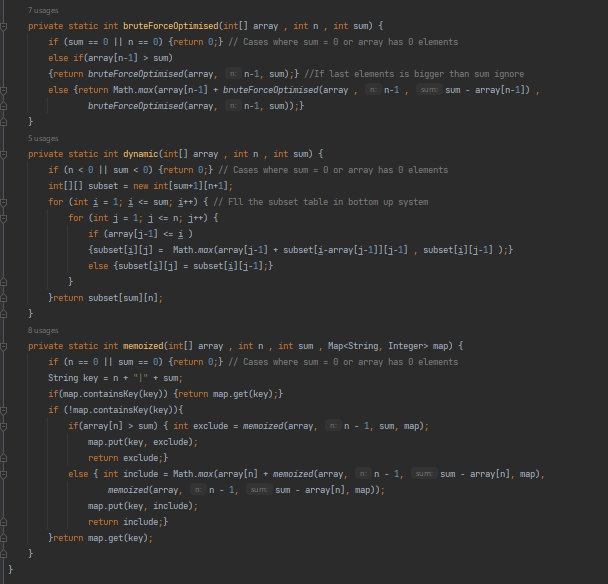


Figure - Task A c) – Alternative Optimisations - Functions

In the results in Figure 13 we can see that the initial optimised algorithm has been functioning quite well until the gap from twenty array elements to thirty where the processing time increased rapidly, thus has been eliminated from the following run as the “Out of Memory Error” appeared. Dynamic programming has been handling low number of array elements well but appears to be getting increasingly better with larger amounts that the previous optimisation as at thirty array elements the optimisation is slowed down massively while dynamic optimisation still handles similar processing time.

The memorised optimisation on the other hand Has done a record time for most array lengths, giving the same result of five array elements and to three thousand array elements. The time complexity for memorised algorithm is O (), where “n” is the size of the sub sum array and “k” is the sum of the elements of the array, very similar to the dynamic time complexity, difference being the use of maps and if statements.

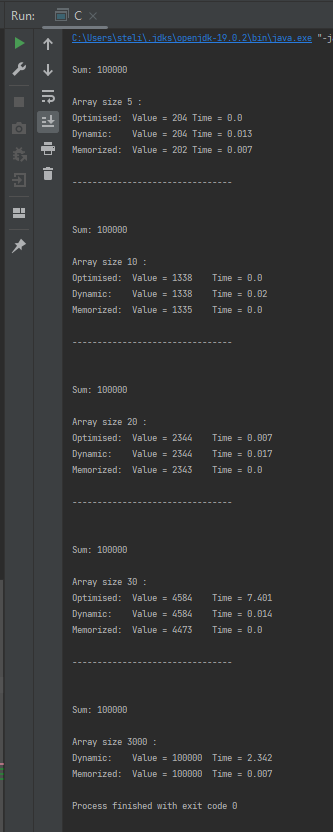


Figure - Task A c) - Alternative Optimisations - Results

# Task B – Filial-heir Chains

## Formula 1 – Number of Edges

If there are nodes in a perfect 3-ary tree (), the number of edges in the equivalent filial-heir chain, is given by:

Demonstration:

|  |  |  |  |
| --- | --- | --- | --- |
| Height | 3-ary tree | Filial-heir chain | Formula |
| Base case  h = 1 |  |  |  |
| h = 2 |  |  |  |
| h = 3 |  |  |  |

This has been proven for the above perfect 3-ary trees of height 1, 2 and 3. Below can be seen a demonstration on a 5-ary full tree, where the formula results in the correct about of edges as expected.

|  |  |  |  |
| --- | --- | --- | --- |
| Height | 5-ary full tree | Filial-heir chain | Formula |
| Base case  h = 1 | A | A |  |
| h = 2 | A    B C D E F | A  B  C  D  E  F |  |
| h = 4 | A    B C D E F  G H I J K L M N O P    Q R S T U | A  B  G C  Q H D  R I E  S J F  T K L  U M  N  O  P |  |

## Formula 2 – Number of total nodes

If there are internal nodes or leaf nodes in a full 5-ary tree (), the total number of nodes, is given by:

OR

Demonstration:

|  |  |  |  |
| --- | --- | --- | --- |
| Height | 5-ary full tree | Filial-heir chain | Formula |
| Base case  h = 1 | A | A |  |
| h = 2 | A    B C D E F | A  B  C  D  E  F |  |
| h = 4 | A    B C D E F  G H I J K L M N O P    Q R S T U | A  B  G C  Q H D  R I E  S J F  T K L  U M  N  O  P |  |
| h = 5 | A    B C D E F  G H I J K L M N O P    Q R S T U    V W X Y Z | A  B  G C  Q H D  V R I E  W S J F  X T K L  Y U M  Z N  O  P |  |

We can observe this does not apply to filial-heir chain trees as they are unbalanced binary trees.

## Formula 3 – Number of leaf nodes

We have had as an example the formula for calculating the amount of leaf nodes on perfect k-ary treats and filial-heir chains, but this formula does not apply to non-perfect tree.

If there are internal nodes or nodes in a full 5-ary tree () or , the number of leaf nodes, is given by:

OR

Demonstration:

|  |  |  |  |
| --- | --- | --- | --- |
| Height | 5-ary full tree | Filial-heir chain | Formula |
| Base case  h = 1 | A | A |  |
| h = 2 | A    B C D E F | A  B  C  D  E  F |  |
| h = 4 | A    B C D E F  G H I J K L M N O P    Q R S T U | A  B  G C  Q H D  R I E  S J F  T K L  U M  N  O  P |  |
| h = 5 | A    B C D E F  G H I J K L M N O P    Q R S T U    V W X Y Z | A  B  G C  Q H D  V R I E  W S J F  X T K L  Y U M  Z N  O  P |  |

This formula can result the amount of leaf nodes only for full trees and not on complete trees or filial-heir chains.

## Formula 4 – Number of internal nodes

If there are nodes or leaf nodes in a full 5-ary tree (), the number of leaf nodes, is given by:

OR

Demonstration:

|  |  |  |  |
| --- | --- | --- | --- |
| Height | 5-ary full tree | Filial-heir chain | Formula |
| Base case  h = 1 | A | A |  |
| h = 2 | A    B C D E F | A  B  C  D  E  F |  |
| h = 4 | A    B C D E F  G H I J K L M N O P    Q R S T U | A  B  G C  Q H D  R I E  S J F  T K L  U M  N  O  P |  |
| h = 5 | A    B C D E F  G H I J K L M N O P    Q R S T U    V W X Y Z | A  B  G C  Q H D  V R I E  W S J F  X T K L  Y U M  Z N  O  P |  |

This formula thus results the expected number of internal nodes, but the formula does not work on complete trees or filial-heir chains.

## Formula 5 – Min and Max height

If there are nodes in a full 5-ary tree (), the min and max height of the tree, and are given by:

AND

Demonstration:

|  |  |  |  |
| --- | --- | --- | --- |
| Height | 5-ary full tree | Filial-heir chain | Formula |
| Base case  h = 1 | A | A |  |
| h = 2 | A    B C D E F | A  B  C  D  E  F |  |
| h = 4 | A    B C D E F  G H I J K L M N O P    Q R S T U | A  B  G C  Q H D  R I E  S J F  T K L  U M  N  O  P |  |
| h = 5 | A    B C D E F  G H I J K L M N O P    Q R S T U    V W X Y Z | A  B  G C  Q H D  V R I E  W S J F  X T K L  Y U M  Z N  O  P |  |

This formula thus results the expected number for the minimum and maximum height, for any tree structure. The only hiccup observed is at 21 nodes as the minimum height resulted 2, but a 5- ary tree of 21 elements cannot be created at a min height = 2. The result of the logarithm in base 5 was quite close to becoming a fraction over 2 which would have been ceiling – ed to 3.

# Appendix

* Task A a) code:
* public class A { // Subset Sum problem - Task A point a)  
   private static boolean isSubsetSum(int[] array , int n , int sum) {  
   if (sum == 0) {return true;} // If sum = 0 return true  
   if (n == 0) {return false;} // If number of array elements = 0 return false  
   else if(array[n-1] > sum) {return *isSubsetSum*(array, n-1, sum);} //If last elements is bigger than sum ignore  
   else {return *isSubsetSum*(array, n - 1, sum) || *isSubsetSum*(array, n - 1, sum - array[n - 1]);}  
   }  
   public static void main(String[] args) // Main function  
   { int[] subset = { 2, 5, 8, 12, 16,45,78,98,52 };  
   int sum = 10;  
   if (*isSubsetSum*(subset, subset.length, sum))  
   {System.*out*.println("TRUE");}  
   else {System.*out*.println("FALSE");}  
   }  
  }
* Task A b) code:
* class B { // Subset Sum problem - Task A point b)  
   static long *initialTime*;  
   static long *lastTime*;  
   static double *time*;  
   private static int isSubsetSum(int[] array , int n , int sum) {  
   if (sum == 0 || n == 0) {return 0;} // Cases where sum = 0 or array has 0 elements  
   else if(array[n-1] > sum)  
   {return *isSubsetSum*(array, n-1, sum);} //If last elements is bigger than sum ignore  
   else {return Math.*max*(array[n-1] + *isSubsetSum*(array , n-1 , sum - array[n-1]) , *isSubsetSum*(array, n-1, sum));}}  
   public static void main(String[] args) {  
   int[] subset = { 2, 5, 8, 12, 16,45,78,98,52 };  
   int sum = 21;  
   System.*out*.println("\nSum: " + sum ); // Display sum  
   System.*out*.println("Array size: " + subset.length ); // Display array length  
   *initialTime* = System.*currentTimeMillis*(); //Start timer  
   int result = *isSubsetSum*(subset, subset.length, sum); // Start function, iterate through all possibilities  
   *lastTime* = System.*currentTimeMillis*(); // Stop timer once function is run  
   *time* = (double) (*lastTime* - *initialTime*) / 1000; // Convert time  
   System.*out*.println("\n\nResults \t" + "\nValue OR Smallest available difference = " + result + "\t\t\t" + "Time = " + *time*);  
   // Print results of either value or smallest available difference plus time recorded  
   }  
  }
* Task A c) code:
* import java.util.HashMap;  
  import java.util.Map;  
  import java.util.Random;  
  public class C{  
   static long *initialTime*; // Time when timer starts  
   static long *lastTime*; // Time when timer stops  
   static double *time*; // Displayed time  
   static int *topSum* = 0; // Counter for set5 sum  
   static Map<String, Integer> *lookup* = new HashMap<>();  
   public static void main(String[] args) { //5 sets or arrays  
   int[] set = {2, 45, 8, 95, 54};  
   int[] set2 = {3, 74, 14, 12, 367, 45, 595, 52, 91, 85};  
   int[] set3 = {1, 93, 5, 10, 515, 40, 4, 50, 68, 43, 28, 356, 24, 123, 82, 456, 58, 71, 6, 311};  
   int[] set4 = {111, 76, 123, 456, 76, 98, 480, 16, 34, 526, 43, 1, 10, 13, 11, 45, 23, 15, 7, 56, 734, 26, 87,  
   98, 12, 237, 5, 10, 515, 640,};  
   int[] set5 = new int[3000];  
   for(int i=0; i<set5.length-1; i++){ //generating random ints for the 5th array  
   Random rand = new Random();  
   set5[i] = rand.nextInt(101);  
   *topSum* += set5[i]; // Counting sum of all elements as they get created  
   }  
   int sum = 100000; // Aimed sum  
   System.*out*.println("\nSum: " +sum + "\n"); // Display aimed sum  
   System.*out*.println("Array size " + set.length + " : "); // Display array size  
    
   *initialTime*= System.*currentTimeMillis*(); // Counter start  
   int result = *bruteForceOptimised*(set, set.length, sum); // Run function  
   *lastTime* = System.*currentTimeMillis*(); // Counter stop  
   *time* = (double)(*lastTime* - *initialTime*)/1000; // Convert time  
   System.*out*.println("Optimised: \t" + "Value = " + result + "\t" + "Time = " + *time* ); // Display result  
    
   *initialTime*= System.*currentTimeMillis*(); // Counter start  
   result = *dynamic*(set, set.length,sum); // Run function  
   *lastTime* = System.*currentTimeMillis*(); // Counter stop  
   *time* = (double)(*lastTime* - *initialTime*)/1000; // Convert time  
   System.*out*.println("Dynamic: \t" + "Value = " + result + "\t" + "Time = " + *time* ); // Display result  
    
   *initialTime*= System.*currentTimeMillis*(); // Counter start  
   result = *memoized*(set, set.length-1, sum , *lookup*); // Run function  
   *lastTime* = System.*currentTimeMillis*(); // Counter stop  
   *time* = (double)(*lastTime* - *initialTime*)/1000; // Convert time  
   System.*out*.println("Memorized: \t" + "Value = " + result + "\t" + "Time = " + *time* ); // Display result  
    
   System.*out*.println("\n--------------------------------\n"); // Separating iterations  
   System.*out*.println("\nSum: " +sum + "\n"); // Display aimed sum  
   System.*out*.println("Array size " + set2.length + " : "); // Display array size  
    
   *initialTime*= System.*currentTimeMillis*(); // Counter start  
   result = *bruteForceOptimised*(set2, set2.length, sum); // Run function  
   *lastTime* = System.*currentTimeMillis*(); // Counter stop  
   *time* = (double)(*lastTime* - *initialTime*)/1000; // Convert time  
   System.*out*.println("Optimised: \t" + "Value = " + result + "\t" + "Time = " + *time* ); // Display result  
    
   *initialTime*= System.*currentTimeMillis*(); // Counter start  
   result = *dynamic*(set2, set2.length,sum); // Run function  
   *lastTime* = System.*currentTimeMillis*(); // Counter stop  
   *time* = (double)(*lastTime* - *initialTime*)/1000; // Convert time  
   System.*out*.println("Dynamic: \t" + "Value = " + result + "\t" + "Time = " + *time* ); // Display result  
    
   *initialTime*= System.*currentTimeMillis*(); // Counter start  
   result = *memoized*(set2, set2.length-1, sum , *lookup*); // Run function  
   *lastTime* = System.*currentTimeMillis*(); // Counter stop  
   *time* = (double)(*lastTime* - *initialTime*)/1000; // Convert time  
   System.*out*.println("Memorized: \t" + "Value = " + result + "\t" + "Time = " + *time* ); // Display result  
    
   System.*out*.println("\n--------------------------------\n"); // Separating iterations  
   System.*out*.println("\nSum: " +sum + "\n"); // Display aimed sum  
   System.*out*.println("Array size " + set3.length + " : "); // Display array size  
    
   *initialTime*= System.*currentTimeMillis*(); // Counter start  
   result = *bruteForceOptimised*(set3, set3.length, sum); // Run function  
   *lastTime* = System.*currentTimeMillis*(); // Counter stop  
   *time* = (double)(*lastTime* - *initialTime*)/1000; // Convert time  
   System.*out*.println("Optimised: \t" + "Value = " + result + "\t" + "Time = " + *time* ); // Display result  
    
   *initialTime*= System.*currentTimeMillis*(); // Counter start  
   result = *dynamic*(set3, set3.length,sum); // Run function  
   *lastTime* = System.*currentTimeMillis*(); // Counter stop  
   *time* = (double)(*lastTime* - *initialTime*)/1000; // Convert time  
   System.*out*.println("Dynamic: \t" + "Value = " + result + "\t" + "Time = " + *time* ); // Display result  
    
   *initialTime*= System.*currentTimeMillis*(); // Counter start  
   result = *memoized*(set3, set3.length-1, sum , *lookup*); // Run function  
   *lastTime* = System.*currentTimeMillis*(); // Counter stop  
   *time* = (double)(*lastTime* - *initialTime*)/1000; // Convert time  
   System.*out*.println("Memorized: \t" + "Value = " + result + "\t" + "Time = " + *time* ); // Display result  
    
   System.*out*.println("\n--------------------------------\n"); // Separating iterations  
   System.*out*.println("\nSum: " +sum + "\n"); // Display aimed sum  
   System.*out*.println("Array size " + set4.length + " : "); // Display array size  
    
   *initialTime*= System.*currentTimeMillis*(); // Counter start  
   result = *bruteForceOptimised*(set4, set4.length, sum); // Run function  
   *lastTime* = System.*currentTimeMillis*(); // Counter stop  
   *time* = (double)(*lastTime* - *initialTime*)/1000; // Convert time  
   System.*out*.println("Optimised: \t" + "Value = " + result + "\t" + "Time = " + *time* ); // Display result  
    
   *initialTime*= System.*currentTimeMillis*(); // Counter start  
   result = *dynamic*(set4, set4.length,sum); // Run function  
   *lastTime* = System.*currentTimeMillis*(); // Counter stop  
   *time* = (double)(*lastTime* - *initialTime*)/1000; // Convert time  
   System.*out*.println("Dynamic: \t" + "Value = " + result + "\t" + "Time = " + *time* ); // Display result  
    
   *initialTime*= System.*currentTimeMillis*(); // Counter start  
   result = *memoized*(set4, set4.length-1, sum , *lookup*); // Run function  
   *lastTime* = System.*currentTimeMillis*(); // Counter stop  
   *time* = (double)(*lastTime* - *initialTime*)/1000; // Convert time  
   System.*out*.println("Memorized: \t" + "Value = " + result + "\t" + "Time = " + *time* ); // Display result  
    
   System.*out*.println("\n--------------------------------\n"); // Separating iterations  
   System.*out*.println("\nSum: " +sum + "\n"); // Display aimed sum  
   System.*out*.println("Array size " + set5.length + " : "); // Display array size  
    
   *initialTime*= System.*currentTimeMillis*(); // Counter start  
   result = *dynamic*(set5, set5.length,sum); // Run function  
   *lastTime* = System.*currentTimeMillis*(); // Counter stop  
   *time* = (double)(*lastTime* - *initialTime*)/1000; // Convert time  
   System.*out*.println("Dynamic: \t" + "Value = " + result + "\t" + "Time = " + *time* ); // Display result  
    
   *initialTime*= System.*currentTimeMillis*(); // Counter start  
   result = *memoized*(set5, set5.length-1, sum , *lookup*); // Run function  
   *lastTime* = System.*currentTimeMillis*(); // Counter stop  
   *time* = (double)(*lastTime* - *initialTime*)/1000; // Convert time  
   System.*out*.println("Memorized: \t" + "Value = " + result + "\t" + "Time = " + *time* ); // Display result  
    
   }  
   private static int bruteForceOptimised(int[] array , int n , int sum) {  
   if (sum == 0 || n == 0) {return 0;} // Cases where sum = 0 or array has 0 elements  
   else if(array[n-1] > sum)  
   {return *bruteForceOptimised*(array, n-1, sum);} //If last elements is bigger than sum ignore  
   else {return Math.*max*(array[n-1] + *bruteForceOptimised*(array , n-1 , sum - array[n-1]) ,  
   *bruteForceOptimised*(array, n-1, sum));}  
   }  
   private static int dynamic(int[] array , int n , int sum) {  
   if (n < 0 || sum < 0) {return 0;} // Cases where sum = 0 or array has 0 elements  
   int[][] subset = new int[sum+1][n+1];  
   for (int i = 1; i <= sum; i++) { // Fll the subset table in bottom up system  
   for (int j = 1; j <= n; j++) {  
   if (array[j-1] <= i )  
   {subset[i][j] = Math.*max*(array[j-1] + subset[i-array[j-1]][j-1] , subset[i][j-1] );}  
   else {subset[i][j] = subset[i][j-1];}  
   }  
   }return subset[sum][n];  
   }  
   private static int memoized(int[] array , int n , int sum , Map<String, Integer> map) {  
   if (n == 0 || sum == 0) {return 0;} // Cases where sum = 0 or array has 0 elements  
   String key = n + "|" + sum;  
   if(map.containsKey(key)) {return map.get(key);}  
   if (!map.containsKey(key)){  
   if(array[n] > sum) { int exclude = *memoized*(array, n - 1, sum, map);  
   map.put(key, exclude);  
   return exclude;}  
   else { int include = Math.*max*(array[n] + *memoized*(array, n - 1, sum - array[n], map),  
   *memoized*(array, n - 1, sum - array[n], map));  
   map.put(key, include);  
   return include;}  
   }return map.get(key);  
   }  
  }

Graphical user interface, text, application

Description automatically generated Table

Description automatically generated with medium confidence

# References

* Biesner, D., Sifa, R. and Bauckhage, C. (2022) ‘Solving Subset Sum Problems using Binary Optimization with Applications in Auditing and Financial Data Analysis’. TechRxiv. Available at: <https://doi.org/10.36227/techrxiv.18994160.v1>.
* Boyles, S.D. (2015) ‘Basic Optimization Concepts’.
* Casselman, B. (no date) ‘Notes on Knuth equivalence’.
* Choudum, S.A. and Lavanya, S. (no date) ‘Complete k-ary trees and Hamming graphs’.
* Koiliaris, K. and Xu, C. (2015) ‘A Faster Pseudopolynomial Time Algorithm for Subset Sum’. arXiv. Available at: <http://arxiv.org/abs/1507.02318> (Accessed: 12 February 2023).
* Leighton, T. and Rubinfeld, R. (2006) ‘Graph Theory III.pdf’. Available at: <http://web.mit.edu/neboat/Public/6.042/graphtheory3.pdf> (Accessed: 16 March 2023).
* Parry, C. (no date) ‘Reduction of SUBSET-SUM-OPTIMIZATION to SUBSET-SUM-DECISION’.
* Pfaltz, J.L. (1975) ‘Representing Graphs by Knuth Trees’, *Journal of the ACM*, 22(3), pp. 361–366. Available at: <https://doi.org/10.1145/321892.321898>.
* Phillips, T. (2021) *How to find all solutions to the SUBSET-SUM problem*, *Medium*. Available at: <https://towardsdatascience.com/how-to-find-all-solutions-to-the-subset-sum-problem-597f77677e45> (Accessed: 10 April 2023).
* Smithline, L. (1995) ‘Bandwidth of the complete k-ary tree’, *Discrete Mathematics*, 142(1–3), pp. 203–212. Available at: <https://doi.org/10.1016/0012-365X(93)E0219-T>.
* ‘Subset Sum problem’ (2011) *GeeksforGeeks*, 24 September. Available at: <https://www.geeksforgeeks.org/subset-sum-problem/> (Accessed: 6 March 2023).
* *Subset Sum Problem Explained (Dynamic Programming)* (no date) *FavTutor*. Available at: <https://favtutor.com/blogs/subset-sum-problem> (Accessed: 7 April 2023).
* *Subset Sum Problems - javatpoint* (no date). Available at: <https://www.javatpoint.com/subset-sum-problems> (Accessed: 23 January 2023).
* *Tree* (no date). Available at: <http://pioneer.netserv.chula.ac.th/~skrung/2301232/Lecture011.html> (Accessed: 9 February 2023).
* *[Tutorial] Subset Sum Square Root Optimisation* (no date) *Codeforces*. Available at: <https://codeforces.com/blog/entry/97396> (Accessed: 1 March 2023).