**24**

**SolveSUBSETSUM** takes as input an array of positive integers **A={a1 ... an}**, and a target integer **T** and returns a subset of **A** whose sum is **T**. If no such subset exists, then it **returns "None"**.

**SUBSETSUM** is the decision version of the above problem.

**Prove the following:**

**(a)** If there is an O(n^k) algorithm B for **SolveSUBSETSUM**, then there is a polynomial time algorithm for **SUBSETSUM**

def SUBSETSUM( A[ 1…n ], t ):

if B(A[ 1…n ] , t ) != None :

return True

return False

**Proof -** Assume that there exists no Subset of a with sum t, so SolveSUBSETSUM or B returns None; it is implied that there is no Subset with a sum equal to t by definition. The result of SUBSETSUM is False.

Assuming that there exists a Subset of a with sum t, now SolveSUBSETSUM or B returns something other than None, which by definition suggests the existence of a Subset with a sum of t. SUBSETSUM will therefore return True.

Time Complexity - Complexity of Algorithm B = O( nk )

**(b)** If there is an O(n^k) algorithm C for SUBSETSUM, then there is a polynomial time algorithm for SolveSUBSETSUM.

def SolveSUBSETSUM( A[1…n], t ):

// base case target sum is achieved

if t == 0:

return

// base case if sum equals target is not possible

if C( A[ 1…n], t ) == False:

print(None)

return

// A[ 1 ] element is included

if C( A[ 2…n], t - A[ 1 ] ) == True:

print( A[ 1 ] )

SolveSUBSETSUM( C( A[ 2…n], t - A[ 1 ] )

// A[ 1 ] element is excluded

else:

SolveSUBSETSUM( C( A[ 2…n], t )

**Base Case:**

There are two base cases

**A -** If t==0, which denotes that the necessary sum has been reached, the function should stop, and the program should return.

**B -** If achieving the required sum is not possible with the given array, print None and return.

**Complexity analysis -**

T( 0 ) = 1 //Base Case

**T( n ) = T( n - 1 )** *(as the array A is getting trimmed by one element in each recursive call)*

**+ O( nk  )**  *(SolveSUBSETSUM is a polynomial time algorithm)*

Upon solving the recurrence:

**T( n ) = O( nk+1)**

**Proof of correctness -**

Assume that there isn't a subset of A with a sum of elements equal to t. This indicates that SUBSETSUM will, by definition, return False, and hence the above method **SolveSUBSETSUM** will print None.

Assume that there is a subset of A with the sum of all elements equal to t. So for every element in the array, there are two choices. The design choice is to check for the sum by including/excluding every element.

**Case A:** **An element is included in the subset**

* Hence, reduce the target by that element and again check if the reduced target can be achieved on the reduced array by using algorithm **C** for **SUBSETSUM**.

If **C(A[ 2…n], t - A[ 1 ]) == TRUE**, then print that element & recursively check for *SolveSUBSETSUM* with the reduced target on the reduced array.

**Case B: An element is not included in the subset**

* Recursively check for *SolveSUBSETSUM* with the same target on the reduced array.

**(c)** **OPTSUBSETSUM**: Given inputs A and T, find and return the largest t <= T such that A has a subset whose sum is t.

def OPTSUBSETSUM( A, t ):

for i in 0 to floor( log t ):

A = A + { 2i }

*// add values 20, 21, 22, 23, … , 2log(t) to the set A.*

for i in floor( log t ) to 0:

A = A - { 2i }

if SUBSETSUM( A, t ) == False:

t = t - 2i

*// remove each power of 2 from A & check if SUBSETSUM(A,t) returns False.*

return t

**Proof of Correctness:**

**Lemma:***Let P represents the original elements of A & Q represents the set of powers of 2 (as defined above), then* ***A = P + Q****. If SUBSETSUM( P+Q , t ) == False, then there exists no subset in P which can generate a number greater than ( t - 2log t ).*

**Proof:**

Let's assume that SUBSETSUM( P+Q, t) == False, and a subset exists in P, whose sum of elements is greater than ( t - 2log t). Let S be that subset and X be that maximum sum, hence X > t - 2log t . Notice that X must be <= t.

Observe that any subset of Q i.e., any subset of **{ 20, 21… 2log( t )-1 }**, can generate any sum between **1 to (2log t - 1).**

Let the sum of elements in S that belong to Q be Y. Hence Y <=2log t -1 **--- (1)**

Now, X > t - 2log t => X >= (t - 2log t  +1) [ because we are working with integer values only ] **—(2)**

From **( 1 ) and ( 2 )** on equality :

X + Y = ( t - 2log t  +1 ) + (2log t - 1)

X+Y = t

This implies that there exists a subset in P+Q, whose sum of elements is t. But this contradicts our given fact that SUBSETSUM( P+Q, t)=FALSE.

So, if SUBSETSUM( P+Q, t ) == False, then there cannot exist any subset in A which can generate a subset such that the sum of elements is greater than ( t - 2log t ).

**Explanation** -

1. Now we **remove 2log ( t )** from A and check if there exists a subset of elements such that target t can be achieved, that is SUBSETSUM( A, t ) == True, which implies that there exists a subset of elements in P which sums up to at least ( t - 2log ( t ) + 1) and with the help elements from Q, i.e. { 20, 21… 2log( n )-1 }, the sum can reach up to t.
2. Now, on the other hand, if we **remove 2log ( t )** from A and check if there does not exist a subset of elements with the sum of elements = target t , that is SUBSETSUM(A,t) == False, then according to our lemma, the target will be reduced to (t - 2log t).

Repeat steps 1 or 2 until all the elements from Q are empty, then after the last iteration, t will contain the largest value <= T such that A has a subset whose sum is t

**// Note that the algorithm will return True if there is a subset with sum equal to t.**

**Complexity analysis :**

**Loop - 1:** loop 1 will execute log (t) times, hence complexity of loop1 is O ( log t )

**Loop - 2:** loop 2 will also execute log (t) times. Each iteration will execute SUBSETSUM problem, which takes O( nK ) time. Hence overall time complexity will be O( nk log ( t ) )

**Total Time Complexity - O(log(t)) + O(nk log(t)) = O(nk log (t))**