HW24

(a) algorithm

```
def SubsetSum ( A, T):
    r = B ( A, T)          — (1)
    if ( r is None )       — (2)
        return False       — (3)
    return True.           — (4)
```

O(n^k) — (1)
O(c) — (2)
O(c) — (3)
O(c) — (4)

where A is the array of elements, T is the target integer and B( ) is the algorithm for SolveSubsetSum

time complexity -

let T(n) be the time complexity of SubsetSum( )

then T(n) = Time (B(A,T)) + O(c)

as shown above = $O(n^k) + O(c)$

= $O(n^k)$ which is polynomial time

Justification

SubsetSum (A, T) is a decision problem which returns True if there exist a subset of A with target sum equal to T and if no such subset exist, returns a False. To solve this problem we take the help of B(A,T) which is algorithm for SolveSubsetSum ( ) which returns a subset whose sum is exactly T and if no such subset exist then it prints None. Thus we call r = B(A,T) and check the value of r. If it is none then no such subset was found whose sum

is equal to T and thus we return False else we return True as then x would have printed / contained the required subset

✓ If SolveSubsetSum() returns a ~~YES instn~~ subset of set A then SubsetSum() returns a yes instance

Let SolveSubsetSum(A,T) return a subset of A, thus we know that ~~sum~~ sum of this ~~subset~~ subset is exactly equal to T. So after calling SolveSubsetSum in ① we check if its return type is not none then we return a True for subsetSum(A,T) ie there exist a subset of A whose sum is equal to T

✓ If SolveSubsetSum() does not returns a subset of A OR it returns None then SubsetSum() returns a No instance

Let ~~Solv~~ for the array A and target T, there be no such subset- ~~st~~ whose sum is equal to T, so SolveSubsetSum() will print ~~return a~~ None. Now after calling SolveSubsetSum() on array A and target T we check if x is None then we return a False ie a No instance for SubsetSum() as ~~we know that~~ there exist no such subset of A whose sum is equal to T.

(6) algorithm

```
def SolveSubsetSum ( A{a1, a2, .. an}, T) :
(1) if ( |A| == 1  && T == a1)
(2)         print (a1)
(3)         return.
(4) if ( |A| == 1  && T ≠ a1 )
(5)         return.
O(n^4) —— (6) r = C ( A{a1 a2 .. an}, T)
(7) if ( r == False)
(8)         return
(9) else :
(10)        A' = copy of A
(11)        remove 1st element a1 from A'
O(n^k) —— (12) r' = C(A', T)
(13) if ( r' == True ) :
T(n-1) —— (14)      Solve subset sum ( A'{a2, a3 .. an},
                                T')
(15) else :
(16)        print ( a1 )
T(n-1) —— (17) Solve subset sum ( A'{a2, a3, .. an},
                                T - a1 )
```

time complexity

let T(n) be the time complexity of Solve subset-sum()

then $T(n) = 2 * \text{time}(C(A, T)) + T(n-1) + O(c)$ (constant) as shown above. $O(c)$ is for rest of the constant operations

$\Rightarrow 2 * O(n^4) + T(n-1) + O(c)$ (constant)

$\Rightarrow O(n^{4+1})$ which is polynomial time.

Justification

Solve SubsetSum (A,T) returns a subset of A or prints. whose sum is exactly equal to T. To solve this problem we take the help of C(A,T) which is an algorithm for subset sum which returns True if such a subset of A exist whose sum is equal to T and else False. If array A contains a single element then we can have a subset of A whose sum is equal to T only when that single element is equal to T itself. This we check in lines ① and print that element and return. If that element is not equal to target T then no such subset will exist and hence we return. (in line ⑨). This way we handle the base cases. Now given an array A with target T we check whether there exist a subset whose sum is equal to T,

~~if x retu~~

if we get a False then we simply return as there is no need to check further in the array A as the entire array ~~will not give/print any s~~ does not have any such subset to print whose sum is equal to T. If it does not ie if we get a True (line (9)) we now know there exist a subset of A and thus we have to print that subset of elements whose sum is equal to T. A' contains the entire array A except the 1st element (line (11)). We check that whether without the 1st element of A ~~is~~ it is possible to achieve the target sum T. ~~If yes~~ This we do with the help of C(A, T) decision problem Subset Sum. If ~~it~~ it returns a Yes / True ie it is possible to get a subset whose sum is equal to T, then we can be sure that even without this element we can achieve our ~~target~~ sum T and thus we don't print it (line (13)) as part of our subset and recursively call SolveSubsetSum ( ) for rest of the array elements ~~without~~ with the same target sum T. But if C(A, T) returned a False (15 line) then we can be sure that this removed element must be included as a part of subset ~~sum~~ of A as without this element it is not possible to get

a ~~subs~~ subset of A whose sum is T. Thus we print this element (line (16)) and recursively call SolveSubsetSum( ) but with 1st element removed from A and also as this element is part of our subset we substract it from target as well (T-a1) so that when recursively the next call is made we can check the same thing as described above for A{$a_2 \dots a_n$} with target sum (T-a_1) =T'.

As the function calls are made recursively we will keep on eliminating one element from the array and also printing the required element until we have a single element in A when it will hit the base case ~~of~~ explained earlier.

If SubsetSum returns a Yes instance then Solve-SubsetSum returns a subset of A.

Let SubsetSum return a Yes instance in line (6) (assuming array A contains more than 1 element as if |A|=1 then base cases will handle as explained before) ie True it means that array A contains a subset whose sum is equal to T and thus we will get ~~a~~ that subset. So from line (10) to (17) we check which element will be the part of subset by removing that element and checking if subset is possible or not with same target ~~sum~~, if yes then

that element is not included in the subset and we recursively start checking for the rest of the elements with same target T. But if a subset is not possible ie we get a False then without this element achieving T is not possible hence we print a_i as part of the subset and recursively check for the rest of the elements. Each time for every element we check the possibility of it being a part of the subset. If it is we update T as $T - a_i$ ie now we want to check if with rest of the elements $(T - a_i)$ target is achievable or not. Thus if in line (6) we get a Yes instance we will get a subset of elements as these elements are adding up to give T and whose absence will cause s_{i1} at line (12) to give False and hence those elements will be printed & we will get a subset

✓ If Subset-Sum returns a No instance then Solve-Subset-Sum does not print anything / any subset

Let SubsetSum returns a No instance ie in line (6) we get a False which means there exist no such subset whose sum will be equal to T. Thus we simply return as there is no need to recursively traverse the array with same or updated target

which will eventually print nothing as no elements will add up to T. We are assuming array A contains more than one element as with |A|=1, it will be handled by the base cases.

(c) algorithm

```
def OPTSUBSETSUM(A, T):
    k = floor(log2 T)                          O(⌊log2 T⌋)
    for (i = k to 0)
        add 2^i as an element in A

    t = T
    for (i = k to 0)                           O(⌊log2 T⌋)
        remove 2^i element from A
        if (c(A, t) == True)                   O(n^k)
            continue                           (given in question)
        else
            t = t - 2^i

    return t.
```

Analysis

OPTSUBSETSUM(A, T) returns the largest $t <= T$ such that A has a subset whose sum is equal to t. To achieve this we needed to check if a subset is possible whose sum is equal to any of the values from T to 1, decremented by 1. At any point as at any t if a subset is formed with sum t then that t would be returned. But the complexity would be $O(n^4 \times T)$ which is not a polynomial time algorithm. ($O(n^4)$ is the time complexity for algorithm C, to calculate subset sum). Thus to avoid this we take a different approach and apply the concept of binary search to select a particular set of elements rather than checking for all elements/targets from T to 1. we add 2^i (where $i = \lfloor \log_2 T \rfloor$ to 0) as an element to our existing array A. Now with these set of elements only, we can achieve all the targets, T to 1 (decremented by 1) Thus rather than checking/adding all numbers from T to 1 we add only these set of numbers to our array A. This is done in the 1st for loop. The newly added elements will always give all

the target sum from T to 1 (decremented by 1) as mentioned above. We add these elements so that we can remove one at time and check if my target T is achievable or not. ~~because if at any~~ If achievable it can be due to the array element alone or with the help of existing newly added elements. Then we ~~loop 0~~ run the 2nd for loop ~~right till~~ uptill all elements (newly added in above for loop) are removed to check if it is the array elements only which is contributing to t. At any point after removal of an element if we get a False from C(A,t) then we know that 2^i (the ~~temo~~ removed element) contributes to getting the target t and thus accordingly we ~~upo~~ update our target (explained below).

Now we set $t = T$, where t = largest $t \leq T$ which we will try to achieve. Now at each iteration of the 2nd for loop we will remove one element which are added earlier. and then perform SubsetSum with the help of C algorithm with A as array and t as target sum (initially $t=T$) ~~Ato~~ If ~~subset~~ we can achieve the target t ie if there exist a subset with sum as t without the ~~count~~ current 2^i element, then it means this removed element does not add/contribute to the target sum t

Thus even without it we got a subset with target sum as t. So we dont do any operation and simply continue with the for loop's iteration. We dont change anyway in the target t as it ~~might/ might not~~ be achievable in the further iterations. If our C algorithm gives false (ie the else part of our code) it means we did not get a subset of A whose sum is equal to t. ~~Thus~~ This shows that the removed element must be present to achieve a target sum of t because as mentioned earlier all the newly added elements will definitely ~~not~~ sum to target values from T to 1 so t ~~reason~~ must be achievable. So we have to subtract the removed elements value from ~~so~~ target t cause if we dont do so then the current t will never be achievable and we will get a false for CC) in every iteration. Thus now our new target becomes ~~t' = t~~ $(t-2^i)$ which we see in further iterations is achievable or not ie weather target sum $t-2^i$ is possible or not for a subset of A. In the last iteration of the 2nd for loop all the newly added elements are removed and with the current t, target if we get a subset then we return it else we have our new target as $t-2^i$ where 2^i is the last newly added element. ~~which~~

time complexity

as shown in the algorithm above, T(n) = time complexity of OPTSUBSETSUM is —

$$T(n) = O(\lfloor \log_2 T \rfloor) + O(n^k \lfloor \log_2 T \rfloor)$$

$$\approx O(n^k \times \lfloor \log_2 T \rfloor)$$

which is polynomial time where T is the target given