Experiment 2

Subset Sum Problem

Objective

Subset Sum Problem – For a given array of size n, does there exist a subset that adds up to some target T? Write the best possible algorithm to solve this problem. Analyse it.

Assumption

Array of size n consists of non-negative integers.

Functions (purposes and complexities)

Function name	Purpose	Time Complexity
subsetSum(vector[0…size], targetSum)	Prints all subsets of *vector[0…size]* whose elements add up to the *targetSum*	Best case: Ω (size^2) Worst case: O(size*2^{size})
clear(vector[0…size])	Sets all elements of vector[0…size] to 0	O(size)
nextFixedSizeSubset(subsetSize, setSize, reset)	Returns array *pattern[0…setSize]* which shows pattern for elements in next subset of size *subsetSize* using 0s and 1s if *reset* is 0, otherwise it returns pattern for the first subset of size *subsetSize*. Once function returns pattern for the last subset of size *subsetSize* then next immediate call with same *subsetSize* with *reset* 0 will return NULL. If *subsetSize* changes then function return the pattern for the first subset of the *subsetSize* irrespective of *reset* value. e.g. nextFixedSizeSubset(3, 5, 0) will return array [1 1 1 0 0]. The next immediate call as nextFixedSizeSubset(3, 5, 0) will return array [1 1 0 1 0]. The next immediate call as nextFixedSizeSubset(4, 5, 0) will return array [1 1 1 1 0]. The next immediate call as nextFixedSizeSubset(4, 5, 0) will return array [1 1 1 0 1]. The next immediate call as nextFixedSizeSubset(5, 5, 0) will return array [1 1 1 1 1]. The next immediate call as nextFixedSizeSubset(5, 5, 0) will return NULL.	O(setSize)

patternSum(vector[0…size], pattern[0…size])	Returns sum of all elements of subset of *vector[0…size]* where subset is defined by *pattern[0…size]*.	O(size)
patternCopy(targetVector[0…size], sourceVector[0…size], pattern[0…size], flags[0…size], &tempIndex)	Function will copy elements of subset of *sourceVector[0…size] to targetVector[0…size]* where subset is defined by *pattern[0…size]* and *flag[0…size]* holds flag for each element in *sourceVector[0…size]* to indicate whether the element is already present in *targetVector[0…size]* or not. *tempIndex* holds value of index of last element of *targetVector[0…size]*.	O(size)
patternPrint(vector[0…size], pattern[0…size])	Function prints the subset *of vector[0…size]* on screen where subset is defined by *pattern[0…size]*.	O(size)
copy(targetVector[0…size], sourceVector[0…size])	Copies *sourceVector[0…size]* to *targetVector[0…size]*.	O(size)

Pseudo code

```
Function subsetSum( vector[0…size], targetSum )
    Sum = 0
    newSize = size
    for subsetSize in 1, 2, …, newSize
        clear( flags, newSize )
        tempIndex = 0
        while( pattern[0…newSize] = nextFixedSizeSubset( subsetSize, newSize, 0 ) == NULL )
            sum = patternSum( vector[0…newSize], pattern[0…newSize] )
                if sum <= targetSum then
                    patternCopy( tempVector[0…newSize],
                                 vector[0…newSize],
                                 patter[0…newSize],
                                 flags[0…newSize],
                                 &tempIndex )
                if sum == targetSum then
                    patternPrint( vector[0…newSize], pattern[0…newSize] )
        newSize = tempIndex
        copy( vector[0…newSize], tempVector[0…newSize] );
```

Tracing for input array [1 3 2 4 6 7] and targetSum = 6

Iteration 1:

    Vector = [1 3 2 4 6 7]
    subsetSize = 1
    newSize = 6
    Output generated: [6]

Iteration 2:

    Vector = [1 3 2 4 6]
    subsetSize = 2
    newSize = 5

Output generated: [2 4]

Iteration 3:

Vector = [1 3 2 4]

subsetSize = 3

newSize = 4

Output generated: [1 3 2]

Iteration 4:

Vector = [1 3 2]

subsetSize = 4

newSize = 3 (< subsetSize)

Algorithm terminates

Conclusion

From the above algorithm, it is clear that to find the subsets of array whose elements adds up to some target T (>=0), it requires maximum $O(N*2^N)$ time where N is the size of the array. Although the best case of algorithm takes only $\Omega(N^2)$ time.

Quiz

1) Which method for algorithm design best suits this problem?

A) Greedy method for algorithm design mentioned in the experiment best suits the problem as it reduces array size after each iteration removing element which can't be used in any subset of the current size to get target sum.

2) Compare your implementation with other possible implementations.

A) The problem can be implemented using greedy algorithm having exponential complexity and using dynamic-programming algorithm having pseudo-polynomial complexity but it cannot list all subsets whose elements adds up to the target sum.

3) What would be the best and worst cases for this problem?

A) Best case:

When value of all elements of vector is greater than the targetSum value the algorithm checks for all subsets with size 1 and no new elements will be added in new vector as all elements are non-negative numbers.

Worst case:

Worst case occurs when sum of all elements in each subset of vector is less than targetSum. In this case size of new vector remains same as the size of original vector throughout the execution of program.