Backtracking | Set 4 (Subset Sum) - GeeksforGeeks

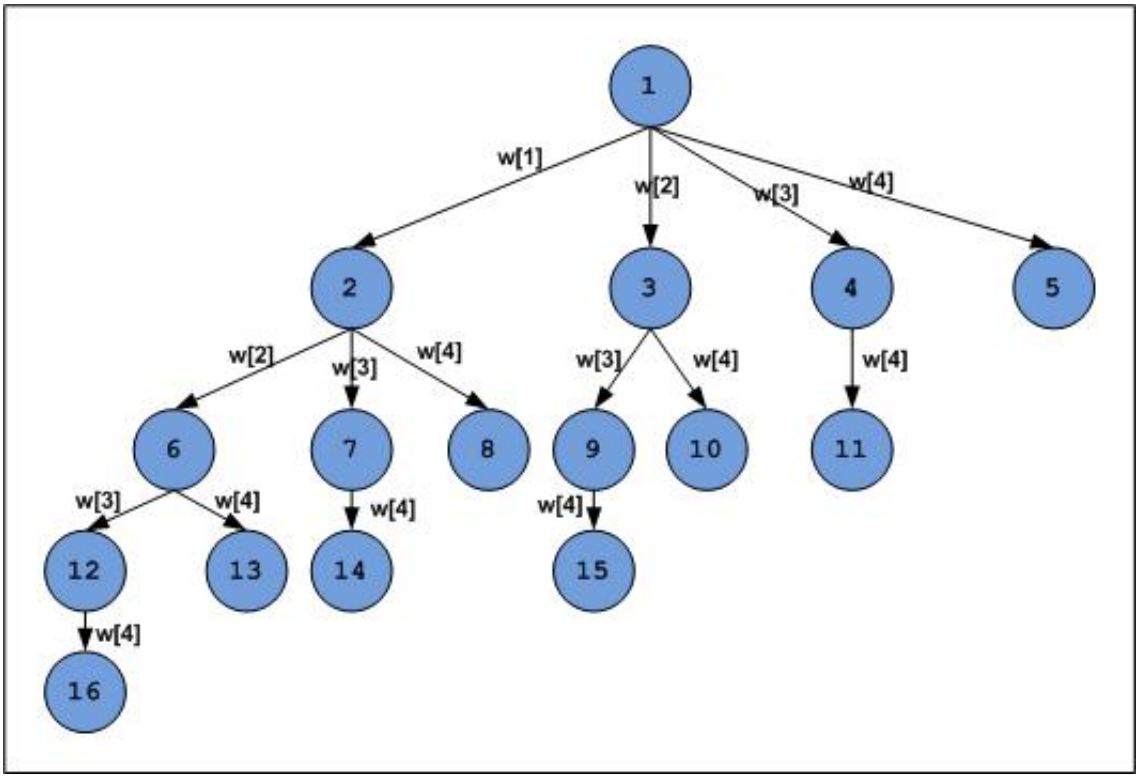
Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

Exhaustive Search Algorithm for Subset Sum

One way to find subsets that sum to K is to consider all possible subsets. A power set contains all those subsets generated from a given set. The size of such a power set is 2^N.

Backtracking Algorithm for Subset Sum
Using exhaustive search we consider all subsets irrespective of whether they satisfy
given constraints or not. Backtracking can be used to make a systematic consideration of the elements to be selected.

Assume given set of 4 elements, say **w[1] … w[4]**. Tree diagrams can be used to design backtracking algorithms. The following tree diagram depicts approach of generating variable sized tuple.



In the above tree, a node represents function call and a branch represents candidate element. The root node contains 4 children. In other words, root considers every element of the set as different branch. The next level sub-trees correspond to the subsets that includes the parent node. The branches at each level represent tuple element to be considered. For example, if we are at level 1, tuple_vector[1] can take any value of four branches generated. If we are at level 2 of left most node, tuple_vector[2] can take any value of three branches generated, and so on…

For example the left most child of root generates all those subsets that include w[1]. Similarly the second child of root generates all those subsets that includes w[2] and excludes w[1].

As we go down along depth of tree we add elements so far, and if the added sum is satisfying explicit constraints, we will continue to generate child nodes further. Whenever the constraints are not met, we stop further generation of sub-trees of that node, and backtrack to previous node to explore the nodes not yet explored. In many scenarios, it saves considerable amount of processing time.

The tree should trigger a clue to implement the backtracking algorithm (try yourself). It prints all those subsets whose sum add up to given number. We need to explore the nodes along the breadth and depth of the tree. Generating nodes along breadth is controlled by loop and nodes along the depth are generated using recursion (post order traversal). Pseudo code given below,

```
if(subset is satisfying the constraint)
    print the subset
    exclude the current element and consider next element
else
    generate the nodes of present level along breadth of tree and
    recur for next levels
```

Following is C implementation of subset sum using variable size tuple vector. Note that the following program explores all possibilities similar to exhaustive search. It is to demonstrate how backtracking can be used. See next code to verify, how we can optimize the backtracking solution.

```
#include <stdio.h>#include <stdlib.h>#define ARRAYSIZE(a)
(sizeof(a))/(sizeof(a[0]))
static int total_nodes;
// prints subset found
void printSubset(int A[], int size)
{
    for(int i = 0; i < size; i++)
    {
        printf("%*d", 5, A[i]);
    }
    printf("\n");
}// inputs// s            - set vector// t            - tuplet vector//
s_size       - set size// t_size       - tuplet size so far// sum
- sum so far// ite          - nodes count// target_sum   - sum to be found
void subset_sum(int s[], int t[],
                int s_size, int t_size,
                int sum, int ite,
                int const target_sum)
{
```

```
    total_nodes++;
    if( target_sum == sum )
    {
        // We found subset
        printSubset(t, t_size);
        // Exclude previously added item and consider next candidate
        subset_sum(s, t, s_size, t_size-1, sum - s[ite], ite + 1,
target_sum);
        return;
    }
    else
    {
        // generate nodes along the breadth
        for( int i = ite; i < s_size; i++ )
        {
            t[t_size] = s[i];
            // consider next level node (along depth)
            subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1,
target_sum);
        }
    }
}// Wrapper to print subsets that sum to target_sum// input is weights
vector and target_sum
void generateSubsets(int s[], int size, int target_sum)
{
    int *tuplet_vector = (int *)malloc(size * sizeof(int));
    subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);
    free(tuplet_vector);
}
int main()
{
    int weights[] = {10, 7, 5, 18, 12, 20, 15};
    int size = ARRAYSIZE(weights);
    generateSubsets(weights, size, 35);
    printf("Nodes generated %d\n", total_nodes);
    return 0;
}
```

Run on IDE

The power of backtracking appears when we combine explicit and implicit constraints, and we stop generating nodes when these checks fail. We can improve the above algorithm by strengthening the constraint checks and presorting the data. By sorting the initial array, we need not to consider rest of the array, once the sum so far is greater than target number. We can backtrack and check other possibilities.

Similarly, assume the array is presorted and we found one subset. We can generate next node excluding the present node only when inclusion of next node satisfies the constraints. Given below is optimized implementation (it prunes the subtree if it is not satisfying contraints).

```
#include <stdio.h>#include <stdlib.h>#define ARRAYSIZE(a)
(sizeof(a))/(sizeof(a[0]))
static int total_nodes;
// prints subset found
void printSubset(int A[], int size)
{
    for(int i = 0; i < size; i++)
    {
        printf("%*d", 5, A[i]);
    }
    printf("\n");
}// qsort compare function
int comparator(const void *pLhs, const void *pRhs)
{
    int *lhs = (int *)pLhs;
    int *rhs = (int *)pRhs;
    return *lhs > *rhs;
}// inputs// s             - set vector// t             - tuplet vector//
s_size       - set size// t_size       - tuplet size so far// sum
- sum so far// ite           - nodes count// target_sum   - sum to be found
void subset_sum(int s[], int t[],
                int s_size, int t_size,
                int sum, int ite,
                int const target_sum)
{
    total_nodes++;
    if( target_sum == sum )
    {
        // We found sum
        printSubset(t, t_size);
        // constraint check
        if( ite + 1 < s_size && sum - s[ite] + s[ite+1] <= target_sum )
        {
            // Exclude previous added item and consider next candidate
            subset_sum(s, t, s_size, t_size-1, sum - s[ite], ite + 1,
target_sum);
        }
        return;
```

```
    }
    else
    {
        // constraint check
        if( ite < s_size && sum + s[ite] <= target_sum )
        {
            // generate nodes along the breadth
            for( int i = ite; i < s_size; i++ )
            {
                t[t_size] = s[i];
                if( sum + s[i] <= target_sum )
                {
                    // consider next level node (along depth)
                    subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1,
target_sum);
                }
            }
        }
    }
}// Wrapper that prints subsets that sum to target_sum
void generateSubsets(int s[], int size, int target_sum)
{
    int *tuplet_vector = (int *)malloc(size * sizeof(int));
    int total = 0;
    // sort the set
    qsort(s, size, sizeof(int), &comparator);
    for( int i = 0; i < size; i++ )
    {
        total += s[i];
    }
    if( s[0] <= target_sum && total >= target_sum )
    {
        subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);
    }
    free(tuplet_vector);
}
int main()
{
    int weights[] = {15, 22, 14, 26, 32, 9, 16, 8};
    int target = 53;
    int size = ARRAYSIZE(weights);
    generateSubsets(weights, size, target);
```

```
    printf("Nodes generated %d\n", total_nodes);
    return 0;
}
```
Run on IDE

As another approach, we can generate the tree in fixed size tuple analogs to binary pattern. We will kill the sub-trees when the constraints are not satisfied.

– – – **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.