14 SUBSET SUM AND 0-1 KNAPSACK PROBLEMS

Problem 1: Subset Sum

The **Subset Sum Problem** is a decision problem. It takes as input a list of integers and is required to determine if a subset of these integers adds up to a given sum *S*.

Sample Input

```
4 6
3 2 7 1
```

Sample Output

YES

Output Details

{3, 2, 1}

Brute Force Solution

Generate all subsets of the numbers and for each subset check if the subset sum is equal to S. Since there are 2^N subset and making subset sum takes linear time, brute force solution runs in $O(2^N N)$ time. This solution may be used only when $N <= 20$.

Recursive Solution with Memoization (DP)

For every number there are two cases: (1) it is in the solution subset, (2) it is not in the solution subset.

- Include the current element in the subset and recur remaining numbers for the remaining sum, or
- Exclude the current element from the subset and recur remaining number for the current sum.
- Return true if any of above recurrence returns true.

$$F(n, S) = \begin{cases} \text{True} & \text{if } n == 0 \text{ and } S == 0 \\ \text{False} & \text{else if } n == 0 \ || \ S < 0 \\ F(n-1, S - A[n]) \ \text{OR} \ F(n-1, S) & \end{cases}$$

This recursive solution is still exponential as it calculates sub-problems over and over again. We can prevent it with memorization technique. For every sum, we may have at most n elements to obtain it. The time complexity and the space complexity of this solution will be $O(NS)$.

Implementation with Recursion and Memoization. *O(NS)* Time and *O(NS)* Space.

```
//Subset Sum with memoization
#include <iostream>
#include <vector>

using namespace std;

vector<int> A = {0, 3, 2, 7, 1}; //0 is a dummy element
vector<vector<int>> dp;
int N, S;
//------------------------------
//Return true if there is a subset having sum S
bool go(int n, int sum)
{
	//A is a 0-based array
	if (n == 0 && sum == 0)
		return true;
	if (n == 0 || sum < 0)
		return false;
	if (dp[n][sum] >= 0)
		return dp[n][sum];

	return dp[n][sum] = go(n - 1, sum - A[n]) || go(n - 1, sum);
}
//------------------------------
int main()
{
	N = A.size();
	S = 6;
	dp.resize(N+1, vector<int>(S+1, -1));

	if (go(N-1, S))
		cout << "YES" << endl;
	else
		cout << "NO" << endl;
}
```

```
Output

YES
```

Implementation with Tabulation. *O(NS)* time and *O(S)* space.

```
// Subset Sum with tabulation
#include <pch.h>
#include <iostream>
#include <vector>

using namespace std;

vector<int> A = { 3, 5, 7, 2 }; //0 is a dummy element
int N, S;
//------------------------------
//Return true if there is a subset having sum S
bool go()
{
	vector<int> dp(S+1, false);
	dp[0] = true;

	for (int i = 0; i < N; i++)
		for (int j = S - A[i]; j >= 0; j--)
			dp[j + A[i]] = dp[j + A[i]] || dp[j];

	return dp[S];
}
//------------------------------
```

```
int main()
{
	N = A.size();
	S = 11;

	if (go())
		cout << "YES" << endl;
	else
		cout << "NO" << endl;
}
```

```
Output

NO
```

Exercise 1: Number of Subsets Having Sum S

Calculate number of subsets of *N* elements that are selected from a given set whose sum adds up to a given number S. Your algorithms should run in *O(NS)* time.

Sample Input

```
5 7
5 2 7 3 4
```

Sample Output

```
3
```

Output Details

```
{5, 2}
{3, 4}
{7}
```

Exercise 2: Printing a Subset Having Sum S

Print one of the subsets of elements that are selected from a given set whose sum adds up to a given number S. Your algorithms should run in *O(NS)* time.

Sample Input

```
5 7
5 2 7 3 4
```

Sample Output

```
{5, 2}
```

Problem 2: Subset Sum Problem with a Large S

Given a set of *N* integers where *N* <= 40. Determine if there exist a subset having sum S where S <= 10^18.

Because S is a very large number we cannot use dynamic programming solution for this problem. On the other hand, we can either not use the standard brute force solution bacuse N is to large for a $O(N2^N)$ time solution.

Solution wiht Meet in the Middle Algorithm

The **Meet in the Middle** algorithm is more efficient, but still exponential, with a time complexity of $0(n2^{N/2})$.

The Meet in the Middle algorithm, splits the input list into equal-size halves. The first half is subject to the same algorithm as the standart brute force solution, in which all subsets are generated, their sums computed, and the sums compared to the target. It is possible but unlikely that the target will be found in the first half. If not, the algorithm generates all subsets of the second half and checks each sum to see if the difference between target and sum was a sum in the first half, in which case the required subset has been found.

You may sort subsets of both halves and use two pointers thecnique or only sort subsets of one halve and make binary search in this halve for every subset of the other halve.

Problem 3: Unbounded Subset Sum Problem

In a classical Subset Sum Problem you may use each item only once, whereas in Unbounded Subset Sum problem repetition of items is allowed.

Unbounded Subset Sum Problem is the same problem with the **Coin Change Problem** where we decide if it is possible to make a payment of S with unlimited number of different denominations of coins.

```
//Unbounded Subset Sum Problem
#include <iostream>
#include <vector>

using namespace std;

vector<int> A = { 3, 5, 7, 2 }; //0 is a dummy element
int N, S;
//------------------------------
//Return true if there is a subset having sum S
bool go()
{
	vector<int> dp(S+1, false);
	dp[0] = true;

	for (int i = 0; i <= S; i++)
		if (dp[i])
			for (int j = 0; j < N; j++)
				if(i + A[j] <= S)
					dp[i + A[j]] = true;

	return dp[S];
}
//------------------------------
int main()
{
	N = A.size();
	S = 11;
```

```
	if (go())
		cout << "YES" << endl;
	else
		cout << "NO" << endl;
}
```

Output

YES

Problem 4: Printing All Subsets Having Sum S

Given an array of integers and a sum S, the task is to print all subsets of given array with sum equal to S.

Sample Input

```
6 10
2 3 5 6 8 10
```

Sample Output

```
5 2 3
2 8
10
```

In this enhanced version of the Subset Sum Problem, you not only need to find if there is a subset with given sum, but also need to print all subsets with given sum.

Solution with DP + Back Tracing

Build a 2D array *dp[][]* such that dp[i][j] stores 0 or number of ways j can obtain with array elements from 1 to *i*.

After filling *dp[][]*, test the last column of dp, if *dp[i][N]* > *dp[i-1][N]*, back trace from that position to print on subset.

Solution with DP + Recursion

Build a 2D array *dp[][]* such that dp[i][j] stores *true* if sum *j* is possible with array elements from 1 to *i*.

After filling *dp[][]*, recursively traverse it from dp[n-1][sum]. For cell being traversed, store path before reaching it and consider two possibilities for the element.

1) Element is included in current path.
2) Element is not included in current path.

Whenever sum becomes 0, stop the recursive calls and print current path.

Problem 5: 0-1 Knapsack Problem

We have a set of *N* items each with an associated weight and value (benefit or profit).

The objective is to fill the knapsack with items such that we have a maximum profit without exceeding the weight limit of the knapsack.

This is a 0-1 Knapsack Problem where we can either take an entire item or reject it completely. We cannot break an item and fill the knapsack.

0-1 Knapsack Problem is a **combinatorial optimization problem**. **Subset Sum problem** is a special case 1-0 Knapsack Problem where items have the same weight and value (*v_i == w_i*).

Sample Input

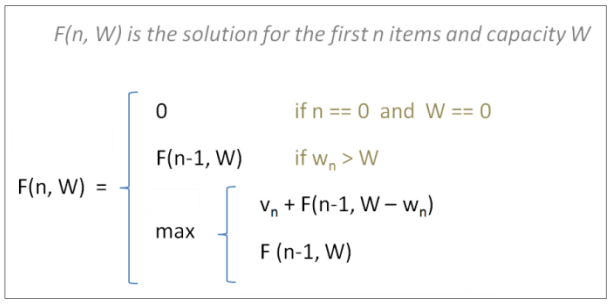
```
4 5
3 4 4 1

100 20 60 40
```

Sample Output

```
140
```

Output Details

```
Item 1 and item 4.
```




```
//0-1 Knapsack
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int N, W;
vector<int> weights = {3, 4, 4, 1};
vector<int> values = { 100, 20, 60, 40 };
//------------------------------
//Return maximum that value can be obtained without crossing
//the limit of the knapsack capacity
int go()
{
	vector<int> dp(W + 1, -1);
	dp[0] = 0;

	//iterate each item
	for (int i = 0; i < N; i++)
	{
		int w = weights[i], v = values[i];
		for (int j = W - w; j >= 0; j--)
```

```
                if (dp[j] >=0)
                      dp[j + w] = max(dp[j + w], dp[j] + v);
      }

      return *max_element(dp.begin(), dp.end());
}
//------------------------------
int main()
{
      N = 4;
      W = 5;

      cout << go() << endl;
}
```

Problem 6: Fractional Knapsack Problem

In the Fractional Knapsack Problem, fractions of items can be taken rather than having to make binary (0-1) choices for each item. Fractional Knapsack Problem can be solvable by **greedy strategy** whereas 0 – 1 Knapsack Problem is not.

Solution with Greedy

1. Compute the value per weight for each item.
2. Take as possible of the item with the highest value per weight.
3. If the supply of that element is exhausted and you can still carry more, take as much as possible of the element with the next value per weight.
4. Sorting, the items by value per weight, the greedy algorithm run in *O(nlogn)* time.

Problem 7: Multidimensional 1-0 Knapsack Problem

The multidimensional 0–1 knapsack problem (MKP) is a special case of general linear 0–1 programs. In this variation, the weight of knapsack item *i* is given by a D-dimensional vector $w_i = \{w_{i1}, w_{i2}, \ldots w_{iD}\}$ and the knapsack has a D-dimensional capacity vector $(W_1, \ldots W_D)$.

The goal is to maximize the sum of the values of the items in the knapsack so that the sum of weights in each dimension *d* does not exceed *Wd*.

You need to enhance the memoization technique of the one dimensional 0-1 knapsack solution to solve the multidimensional 1-0 knapsack problem. You should store the best total value for every possible weight combinations. $dp[w_i][w_j]\ldots[w_m]$ is the best value for the $w_i + w_j + \ldots w_m$ weight. *Dp[0][0]…[0]* is 0.

Useful Links and References

https://www.geeksforgeeks.org/subset-sum-problem-dp-25/
https://www.geeksforgeeks.org/perfect-sum-problem-print-subsets-given-sum/
https://programmingpraxis.com/2012/03/30/subset-sum-meet-in-the-middle/
https://tutorialspoint.dev/algorithm/dynamic-programming-algorithms/perfect-sum-problem-print-subsets-given-sum
https://www.javatpoint.com/fractional-knapsack-problem
https://en.wikipedia.org/wiki/Knapsack_problem#Multi-dimensional_knapsack_problem