10/06/2023

Q1 Partition equal subset sum.

i/p → {1, 5, 11, 5}

o/p → True

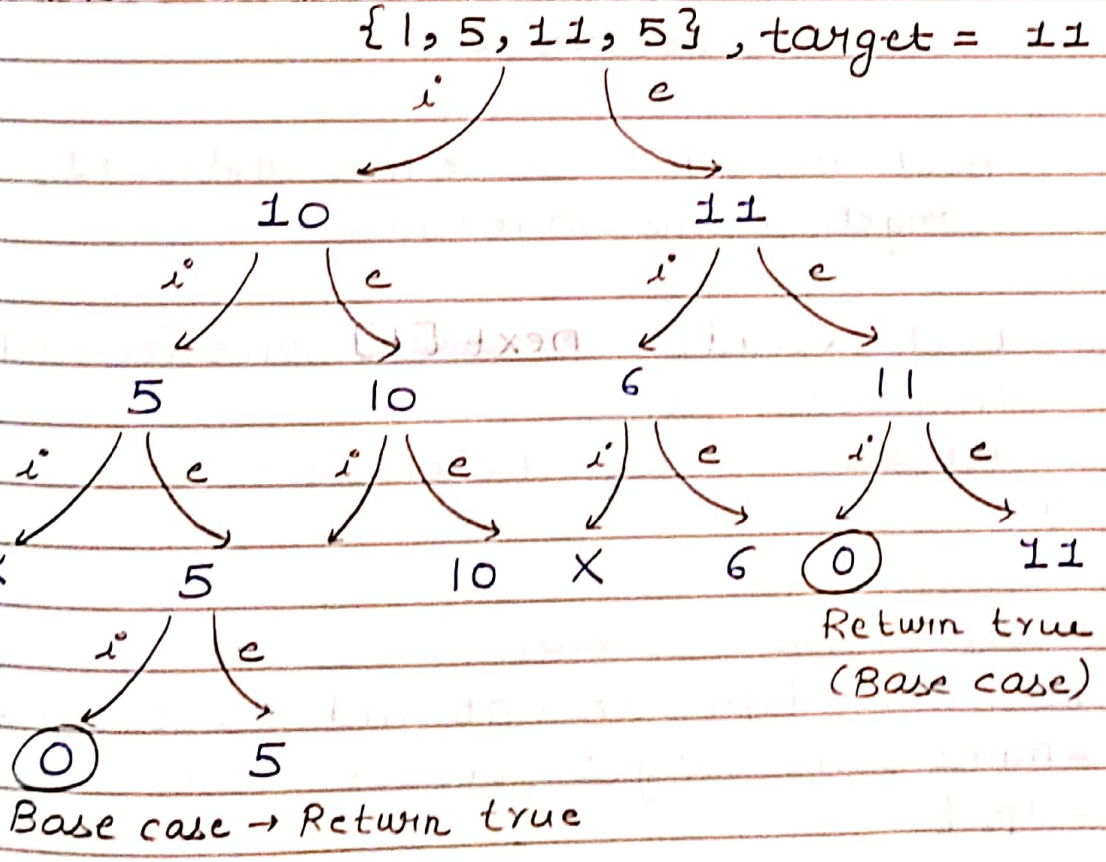
1 + 5 + 11 + 5 = 22

sum = 11

(2) → dividing by 2 as equal subset sum.

Now we need to find the subset having sum 11.

* Include-exclude Pattern



Code

In main function, simply calculate the sum of elements of array. We have to add one case

where if sum is odd, then we have to simply return false. Initialize index = 0, target = sum/2;

```
// Recursive code
bool solveRec(int index, vector<int> &nums,
int target) {

    if (index >= nums.size())
        return 0;
    if (target < 0)
        return 0;
    if (target == 0)
        return 1;

    bool include = solveRec(index+1, nums,
    target - nums[index]);

    bool exclude =          solveRec(index+1,
    nums, target);
    return include || exclude;
}

// Top-down approach
bool solveTopDown(int index, vector<int>
&nums, int target, vector<vector<int>>
&dp) {
        // Base case
        if (index >= nums.size()) {
            return 0;
        }
        if (target < 0)
            return 0;
```

```
if (target == 0)
        return 1;
//Step 3: Check if answer already exists
if (dp[index][target] != -1)
        return dp[index][target];
//include call
bool include = solveTopDown(index+1, nums,
target - nums[index], dp);
//exclude call
bool exclude = solveTopDown(index+1, nums,
target, dp);
//Step 2: Store answer in dp array.
dp[index][target] = include || exclude;
return dp[index][target];
}
//Bottom up approach
bool solveTab(vector<int>& nums, int target)
{
    int n = nums.size();
    //Step 1: Create dp array
    vector<vector<int>> dp(nums.size()+1,
    vector<int>(target+1, 0));
    //Step 2: Observe base case of top down
    for(int i=0; i<nums.size(); i++){
        dp[i][0] = 1;
    }
    //Step 3: Reverse flow of top-down
    for(int index = n-1; index >= 0; index--){
    for(int t = 1; t <= target; t++){
        bool include = 0;
        if(t - nums[index] >= 0)//Valid index check
            include = dp[index+1][t-nums[index]];
```

```
        bool exclude = dp[index+1][t];
        dp[index][t] = include || exclude;
      }
    }
    // Ans is at dp[0][target] as we are going to 0.
    return dp[0][target];
}
```

Note → In top-down approach, simply create 2D vector in main() as
vector<vector<int>> dp(nums.size(), vector<int>(target+1, -1));

Q1 → Why 2D array is used?
As in the recursive call, 2 variables are changing i.e index and target.

Q2 → Space optimization possible or not?
Yes space optimization is possible here, dp[index][t] depends on include and exclude.

include → dp[index+1][t-nums[index]]
exclude → dp[index+1][t]

Hence dp[index][t] depends upon the next row.
Make 2 1D arrays.
vector<int> curr(target+1, 0);
vector<int> next(target+1, 0);

Simply replace dp[index] with curr and

dp[index+1] as next in the bottom up approach.

Also we need to do shifting i.e next = curr as we are going upwards.

Q2 Number of dice rolls with target sum.
Here the pattern that we will be applying is distinct ways in which for loop will be used.

i/p → n = 2 (dices), k = 6 (faces), target = 7

On this test case we will have 36 choices or pairs.

{1,1}	{2,1}	{3,1}	{4,1}	{5,1}	{6,1}
{1,2}	{2,2}	{3,2}	{4,2}	{5,2}	{6,2}
{1,3}	{2,3}	{3,3}	{4,3}	{5,3}	{6,3}
{1,4}	{2,4}	{3,4}	{4,4}	{5,4}	{6,4}
{1,5}	{2,5}	{3,5}	{4,5}	{5,5}	{6,5}
{1,6}	{2,6}	{3,6}	{4,6}	{5,6}	{6,6}

Total ways = 6 in which we can get sum = 7.

Code

```
long long int MOD = 1000000007; given in question
int solveRec(int n, int k, int target) {
    // Base case
    if (n < 0)
        return 0;
```

```
if (n == 0 && target == 0)
        return 1;
if (n == 0 && target != 0)
        return 0;
if (n != 0 && target == 0)
        return 0;
long long int ans = 0;
for (int i = 1; i <= k; i++) {        ↗ target reduced
    ans += solveRec (n-1, k, target - i);
}                          ↳ dice reduced
return ans;
}
// Top down approach
long long int solveTopDown (int n, int k,
int target, vector <vector < long long int >>&
dp) {
    // Base case
    if (n < 0)
        return 0;
    if (n == 0 && target == 0)
        return 1;
    if (n == 0 && target != 0)
        return 0;
    if (n != 0 && target == 0)
        return 0;
    // Step 3 : Check if ans already exists
    if (dp[n][target] != -1)
        return dp[n][target];
    long long int ans = 0;
    for (int i=1; i<=k; i++) {
        long long int recAns = 0;
        if(target - i >=0) // valid index check
```

```
        recAns = solveTopDown(n-1, k, target-i, dp);
    }
    recAns = recAns % MOD;
    ans = ans % MOD;
    ans = (ans + recAns) % MOD;
}
// Step 2 : Store in dp array
dp[n][target] = ans;
return dp[n][target];
}
```

Note → We have created dp array in main() as
vector<vector<long long int>> dp(n+1,
vector<long long int>(target+1, -1));

Why 2D DP is used here?
2D DP was used here as target and n, 2 variables are changing.

```
// Bottom up approach
int solveTab(int n, int k, int target) {
    // Step 1 : Create dp array
    vector<vector<long long int>> dp(n+1,
    vector<long long int>(target+1, 0));
    // Step 2 : Observe base case of top down
    dp[0][0] = 1;
    // Step 3 : Reverse flow of top down
    for(int index = 1; index <= n; index++) {
        for(int t = 1; t <= target; t++) {
            long long int ans = 0;
            for(int i=1; i<=k; i++) {
                long long int recAns = 0;
```

```
        if (t - i >= 0)
            recAns = dp[index-1][t-i];
        ans = (ans % MOD + recAns % MOD) % MOD;
    }
    dp[index][t] = ans;
  }
 }
 return dp[n][target];
}
```

Note → Instead of n, we used index and instead of target, we used t in bottom up code.

Space optimization is possible or not?
dp[index] is dependent upon dp[index-1] i.e previous row & hence space can be optimized.

1) Create 2 1D arrays
vector<int> prev(target+1, 0);
vector<int> curr(target+1, 0);

2) Now instead of dp[0][0] use prev[0].
dp[index] → use curr.
dp[index-1] → use prev.

3) Shifting logic will be prev = curr as we are going downwards.