DP on Subsequence (Subsets)

Subsequence :- Any contiguous or non-contiguous part of array

Ex:- [1, 3, 2] ⇒ [1, 2] ⇒ subsequence (bcz it's non-contagious & following the order of '12')

~~[2, 1]~~ ⇒ has to follow the order.

Contiguous :- Elements are taken next to each other without skipping.

Non-Contiguous :- Elements can be taken by skipping some in between.

Subsequence :- Contiguous or Non-contiguous elements

Subarray :- Only contiguous elements.

Ex:- [1, 3, 2] ⇒ [1, 2], [3, 2] ⇒ subsequence

Subset Sum Equal to K

You are given an array/list 'ARR' of 'N' positive integers & an integer 'K'. Your task is to check if there exists a subset in 'ARR' with a sum equal to 'K'.

Ex:- If 'ARR' is {1, 2, 3, 4} & 'K' = 4, then there exists 2 subsets with sum = 4 ⇒ {1, 3} & {4}

Sol Steps

1) Generate all subsequences & check if any of them gives the sum of K.

How ??

1. PowerSet (Use BitManipulation)
2. Recursion

If you get at least 1 subsequence then your job is done & there is no need to check all subsequences

1) Express in terms of index, target

2) Explore possibilities of that index (2 possibilities for current problem)

a) a[ind] is part of the subsequence

b) a[ind] not part of the subsequence

Our Recurrence :- f(n-1, target)

indicates in the entire array till n-1 does the target exist

Ex:- If f(3,4) ⇒ this checks if from 0-3 does the target exist

f(ind, target){

// Imagine you achieved your target already then you will turn your target into 0 so your base case will return true if your target is 0

a) f(ind, target){
if (target == 0) return true;

2. Imagine you are currently verifying 0th index that indicates you travelled the whole recursion & came to 0th index .. So your condition will verify if ind=0 if your array[ind] == target then return true.

b) if (ind == 0) return (a[0] == target);

(2) Explore Possibilities

f(ind, target) ⇒ Checks if 4 can form a subsequence from 0 to ind if 4 can form a target which has 2 ways

1) bool not take
= f(ind-1, target)

2) bool take = false
why??
if(target >= a[ind]) take = f(ind-1, target-a[ind])

f(ind-1, target-a[ind])

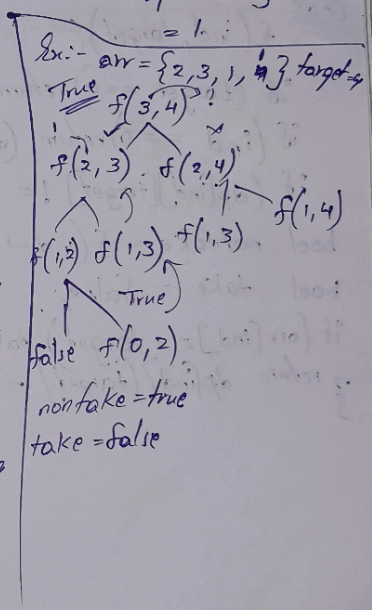
If you choose a[ind]
for example in array {1, 2, 3} k = 4
if you choose ind-1 = 3
then your target will now be to add +1 to get k value
∴ target will become your target value − your chosen value
4 − 3
= 1

return take or not take

Recursion ⟹ TC ⇒ $o(2^n)$
SC ⇒ o(n)

```
f(ind, target) {
  if (target == 0) return true;
  if (ind == 0) return (a[0] == target);
  bool not take = f(ind-1, target);
  bool take = false
    if (target >= a[ind])
        take = f(ind-1, target - a[ind]);
  return (take) or (not take).
```

Ex:- arr = {2, 3, 1, 1} target = 4



We can notice that we are having overlapping sub problems
∴ We'll solve it with memoization

Memoization

TC = O(N × target)
SC = O(N × target) + O(N)

1. Figure out the changing state i.e

ind ↓ target ↓

dp[10^3 + 1] × dp[10^3 + 1]

imagine constraints as target <= 10^3
ind <= 10^3

2. Initialize dp to -1
3. Base case ; if (dp[ind][target] != -1) return dp[ind][target]

Code

```
f(ind, target){
  if (target == 0) return true;
  if (ind == 0) return (arr[0] == target);
  if (dp[ind][target] != -1) return
  bool nottake = f(ind-1, target)
  bool take = false;
  if (arr[ind] <= target) take = f(ind-1, target-arr[ind], arr, dp)
} return dp[ind][target] = take || nottake ;
```

Tabulation

1) dp[n][target] Initialize dp

2) Write base case in the form of dp i.e.;

if (target == 0) return true; ⟹ for (i=0 → n-1){
dp[i][0] = true;

if (ind == 0) return (arr[0] == target) ⟹ dp[0][arr[0]] = true;

3) Form Nested loops ⟹ Tabulation is bottom-up approach therefore our loop goes from 0 → n-1

ind ⟹ (1 → n-1)

target ⟹ (1 → target)

for (int i=0 → n-1) dp[i][0] = true;
dp[0][arr[0]] = true;

```
for (i=1 → n-1){
  for (j=1 → K){
    bool notTake = dp[ind-1][target];
    bool take = false;
    if (arr[ind] <= target)
      take = dp[ind-1][target - arr[ind]]
    dp[ind][target] = take | not Take;
  }
}
return dp[n-1][K];
```

Space Optimization

Instead of using array you store the values in variables. Whenever you call for (i-1), (i-2) or (index-1) we update these variables instead of using array.

Code

```
boolean prev(K+1, 0), cur(K+1, 0);
  prev[0] = curr[0] = true;
  prev[arr[0]] = true;
for(int i=1 → n-1){
  for(j=1 → K){
      bool notTake = prev[target];
      bool take = false;
      if(arr[ind] <= target) take = prev[target - arr[ind]];
      cur[target] = take | notTake;
  }
  prev = cur;
}
return prev[K];
```