09/06/2023

Q1 Painting fence

We would be given some colors and we need to paint the fences such that not more than 2 adjacent fences have the same color.

i/p → X X X X X X X X
R R G G B B R R

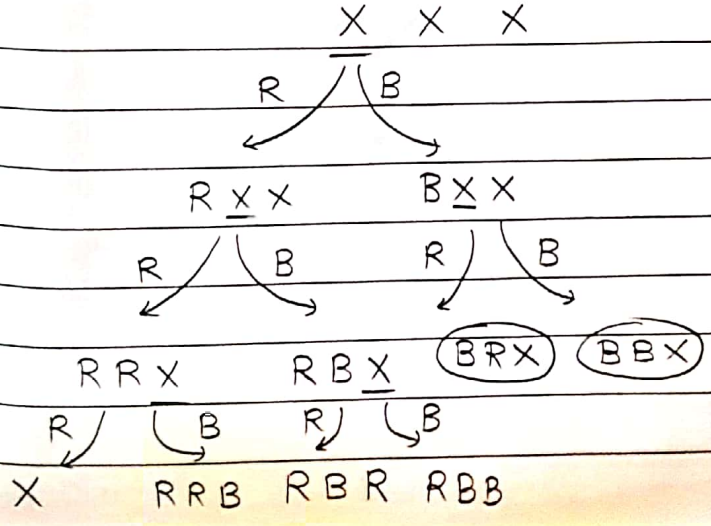
colors = R, G, B

We have to return no. of ways we can color the fence.

n = 3
k = 2

X	X	X
R	R	B
R	B	R
R	B	B
B	R	R
B	R	B
B	B	R

Here there are 6 ways in which we can color the 3 fences with 2 colors.



Scanned with Cam

BRX
R B
BRR BRB

BBX
R B
BBR X

Hence 6 answers are possible.

Ex → Trying to find pattern

$n = 4$, $k = 3$
↳ (R, G, B)

* $n = 1$
same → 0 (not possible)
different → R, G, B (3)

same count

* $n = 2$
same → RR, GG, BB (3)
different → RB, RG, BR, BG, GR, GB (6)

same count

* $n = 3$
same → RBB, RGG, BRR, BGG, GRR, GBB (6)
different → RRB, RRG, BBR, BBG, GGR, GBR, GGB, GBG, RBR, RBG, RGR, RGB, BRB, BRG, BGR, BGB, GRB, GRG (18)

f(4) : XXXX

same → f(2) : RR, GG, BB → f(2) × (k−1)

different → f(3) : R, G, B → f(3) × (k−1)

Scanned with Cam

classmate
321 Date
Page

Why (k-1) is getting multiplied?

Let's assume 1st 2 colors are RR, then we can put only GG and BB i.e 2 choices are there and hence (k-1) choices for each. Similarly for different. Let's assume 1st 3 colors are RGG and then last can be R or B & hence 2 choices which are (k-1) choices.

Recursive relation for n = 4

$f(4) = f(2) \times (k-1) + f(3) \times (k-1)$

$f(4) = [f(2) + f(3)] (k-1)$

Note → $f(n) = [f(n-1) + f(n-2)] \times (k-1)$] generalized

Code

```
int solveRec (int n, int k) {
    // Base case
    if (n == 1)
        return k;
    if (n == 2)
        return k + k * (k-1);
    // Recursive call
    int ans = (solveRec (n-2, k) + solveRec (n-1, k)) *
                                        (k-1);

    return ans;
}
// Top-down approach
int solveTopDown (int n, int k, vector<int> &dp){
    // Base case
    if (n == 1)
        return k;
```

Scanned with Cam

```
    if (n == 2)
        return k + k * (k-1);
    //Step 3 : Check answer already exists
    if (dp[n] != -1)
        return dp[n];
    //Step 2 : Save answer in dp array
    dp[n] = (SolveRec (n-2, k) + SolveRec (n-1, k))
                                        * (k-1);
    return dp[n];
}
//Bottom up approach
int solveTab (int n, int k) {
    //Step 1 : Create dp array
    vector <int> dp (n+1, 0);
    //Step 2 : Observe base case of top-down
    dp[1] = k;
    dp[2] = k + k * (k-1);
    //Step 3 : Reverse flow of top-down
    for (int i = 3; i <= n; i++) {
        dp[i] = (dp[i-2] + dp[i-1]) * (k-1);
    }
    return dp[n];
}
//Space optimization
int spaceOpt (int n, int k) {
    int prev2 = k;
    int prev1 = k + k * (k-1);
    for (int i=3; i<=n; i++) {
        int curr = (prev2 + prev1) * (k-1);
        prev2 = prev1;
        prev1 = curr;
    }
```

Scanned with Cam

```
    return prev1;
}
```

Why space optimization was possible?
In tabulation approach, dp[i] depends upon dp[i-1] and dp[i-2] i.e. previous 2 values only and hence space optimization is possible.

Note → In top-down, dp array was created in main().

2D DP

Q1 Knapsack problem

i/p ⇒ n-items
weight → {w1, w2, w3, w4}
value → {v1, v2, v3, v4}
Max-capacity = W

We need to return the maximum value/profit by not exceeding W.

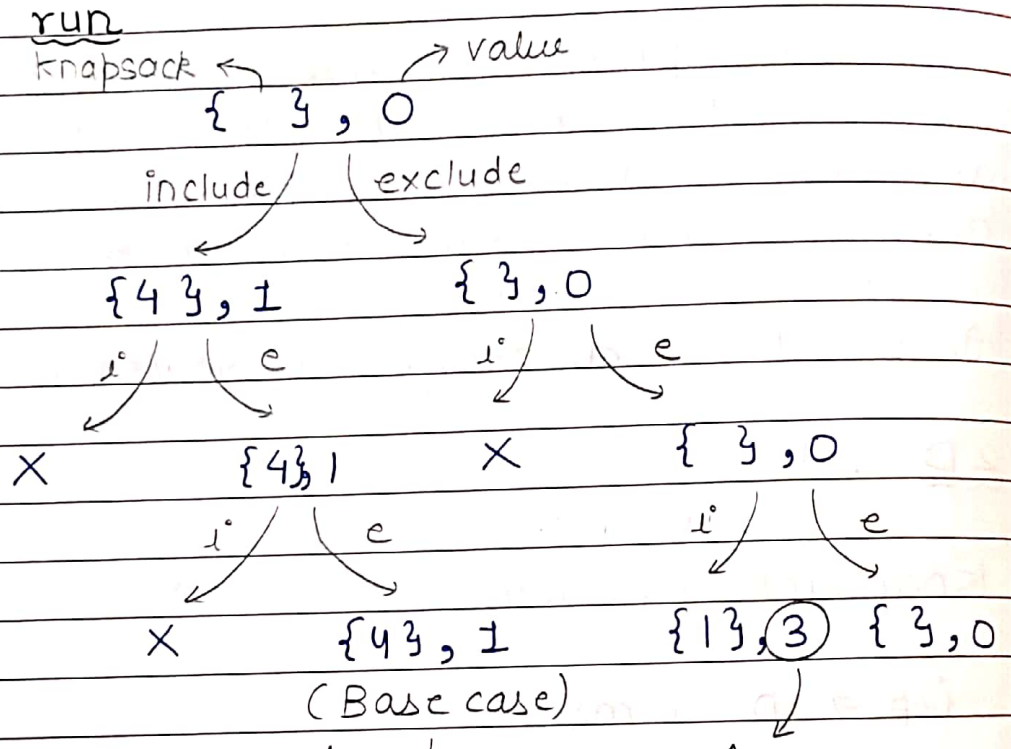
Ex → weight → {4, 5, 1}
value → {1, 2, 3}
capacity = 4

{I} → 1
{II} → x
{III} → (3) ⇒ maximum
{I, III} → x
{I, II} → x
{II, III} → x
{I, II, III} → x

Scanned with Cam

Here include-exclude pattern will happen.

Dry run



Code

```
int solveRec (int weight[], int value[], int
index, int capacity) {
    // Base case → only 1 item
    if(index == 0) {
        if (weight[0] <= capacity)
            return value[0];
        else
            return 0;
    }
    // include call
    int include = 0;
    if(weight[index] <= capacity) {
        include = value[index] + solveRec(
        weight, value, index-1, capacity - weight[index
```

Scanned with Cam

```
    }
    //exclude call
    int exclude = 0 + solveRec (weight, value,
    index-1, capacity);

    return max (include, exclude);
}
// Top-down approach
int solveTopDown (int weight[], int value[],
int index, int capacity, vector<vector <int>>
&dp) {
        //Base case
        if (index == 0) {
            if (weight[0] <= capacity)
                return value[0];
            else
                return 0;
        } //Step 3: Answer already exists
        if (dp[index][capacity] != -1)
            return dp[index][capacity];
        // include call
        int include = 0;
        if (weight[index] <= capacity) {
            include = value[index] + solveTopDown(
            weight, value, index-1, capacity-weight[index]
            ,dp);
        }
        //exclude call
        int exclude = 0 + solveTopDown (weight, value,
        index-1, capacity, dp);
        // Step 2: Store in dp array.
        dp[index][capacity] = max (include, exclude)
    } return dp[index][capacity];
```

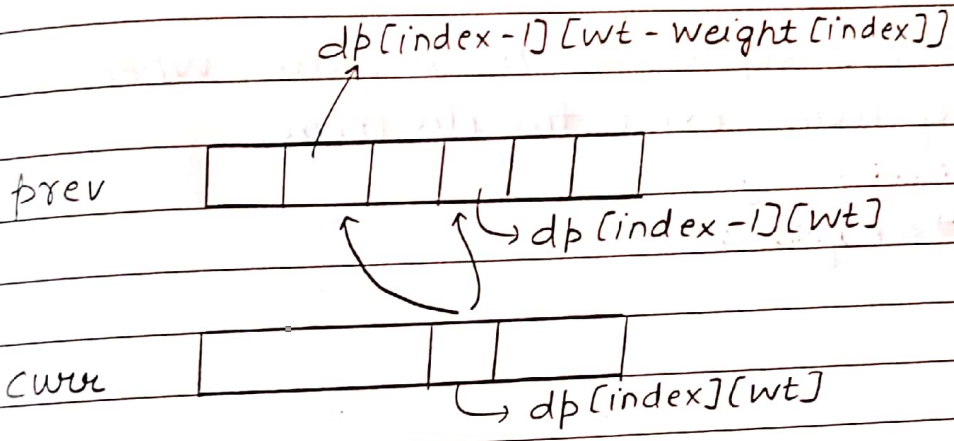
Scanned with CamScanner

Note → vector <vector <int >> dp (n, vector <int> (capacity +1, -1)); → 2D array required in top-down approach

```
// Bottom up approach
int solveTab (int weight [], int value [],
int n, int capacity) {
  //Step 1 : Create dp array
  vector <vector <int >> dp (n, vector <int> (
   capacity +1, 0));
  //Step 2 : Observe base case of top-down.
  Run a loop for 0th row.
  for (int w=0 ; w <= capacity ; w++) {
        if (weight [0] <= capacity)
              dp [0][w] = value [0];
        else
              dp [0][w] =0;
  }
  //Step 3 : Reverse flow of top-down. Replace
  n with index and capacity with wt.
  for (int index = 1; index <n ; index ++) {
     for (int wt = weight [0] ; wt <= capacity ; wt++) {
        // include call
        int include =0 ;
        if (weight [index] <= capacity)
            include = value [index] + dp [index -1][
            wt - weight [index]];
        }
        // exclude call
        int exclude = 0 + dp [index-1] [wt];
        dp[index] [wt] = max (include, exclude);
     }
  }
  return dp[n-1][capacity];
}
```

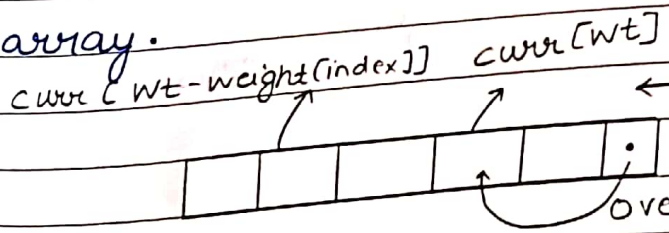
Scanned with Cam

Space optimization possible or not?
dp[index][wt] depends upon include and exclude.
include → dp[index-1][wt-weight[index]];
exclude → dp[index-1][wt]
Hence we can make 2 1D arrays.



1) Make 2 1D arrays
vector<int> prev(capacity+1, 0);
vector<int> curr(capacity+1, 0);
2) Now replace dp[index-1] with prev and dp[index] with curr.

Can we do with the help of single 1D array?
The value which we need to store in curr is dependent upon prev[wt-weight[index]] and not dependent upon the value in prev[wt] and not dependent upon the value in curr array.



overriden value would be considered. To avoid this reverse the flow (Right to Left).

Scanned with Cam

classmate
B28 Date
Page

1) Inner for loop changes

```
for (int wt = capacity ; wt >= 0 ; wt--) {
        ---
}
```

2) Replace prev with curr in space optimized-1 solution.

Note → dp[i][j] represents max value when
→ no. of items exist till ith item
→ capacity = j
↳ knapsack

Scanned with Cam