Day 32 DP2

2 D DP → working with 2 different parameters in parallel.

---

* Recursive : top down approach → additional stack size (only minor disadvantage)
  - ↓
  - When stack calls > 10^5, then problems encountered.
  - ↓
  - generally okay in interviews.

Definitions don't really matter.
↓
Better to think in terms of iterative or recursive

* Iterative : bottom up approach (more efficient
  - ↓
  - no stack calls.
  - ↓
  - no fn calls)

These terms are not really reflective of the complexity of the problems involved in DP

for nth value in a dp based problem, then

$dp[n] = dp[n+1] + dp[n+2]$

↓

values the current parameter is dependent on are already computed.

(pre calculations)

↓

loop in reverse direction

here starting from highest value possible and then moving down.

---

moving from recursive to iterative approach : once the recurrence relation is developed & the dependent smaller problems identified, we can judge which values to compute first & move on from them

(some ordered relation)

Q. Given a grid, starting at (0,0), what are the ways to reach (n,n) given you can move either to the right or to down.

ways to reach n-1, n-1 from i,j (variation: diagonally as well)

↳ ways(i,j) = ways(i+1, j) + ways(i, j+1);

memoize ← (storing value for i, j in a 2D matrix)

base ways(n-1, n-1) = 1

↓

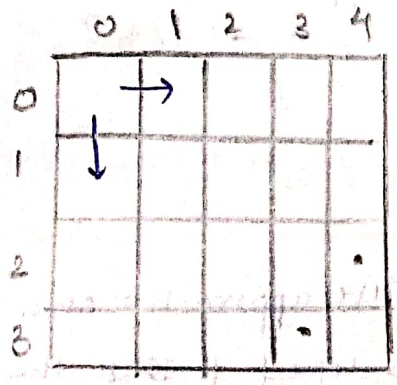
since the current pair of indices depend on the bottom / right index, we need them to be precalculated,

↓

∴ for iterative implementation, start building from (n-1, n-1)

Here the starting coordinates were taken as parameters, but the destination coordinates could also have been used.

ways(N, M) = ways(N-1, M) + ways(N, M-1);



ways(3,4) = ways(2,4)

↓

(that is same smaller problem if standing at (1,0))

+ ways(3,3)

↓

(standing at 0,1)

ways(0,0) = 1

ways(0,i) = ways(0,i-1)

ways(i,0) = ways(i-1,0)

variation: each cell has a value written on it. Based on the value, that many steps can be taken down or to the right. If value = 0, then it is blocked. ways to go from (0,0) to (N,M). If no ways possible, return -1.

↓

0 cells can be skipped (2 0 1)

(memoization) direction 1:

ways(i,j) = ways(i + A[i][j], j) + ways(i, j + A[i][j]);

ways to go from i,j to N,M

where ways(N,M) = 1 if (A[N][M] ≠ 0)
= 0 if (A[N][M] = 0)

↓

along with memoization

↓

(memoization direction 2):

ways(N,N) = ways(N - A[0][0], M) + ways(N, M - A[0][0]);

ways to reach N,N from 1,1

if coordinate < 0, return 0;
if (A[0][0] == 0) return 0;

} base condns

thus check before destination check.

↓ building iteratively.

build from N,M, and applying checks for ways(N,M) = 1, invalid coordinate, we can just add and reach till 0,0.

↓

```
for (i = N; i >= 0; i--)
    for (j = M; j >= 0; j--) {
        if (i >= N, j >= M) dp[i][j] = 1;
        if (A[i][j] >= 0) dp[i][j] = 0;
        else
        dp[i][j] = dp[i + A[i][j]][j] + dp[i][j + A[i][j]]
```

invalid i,j check, ans = 0 if A[i][j]

Q. Edit distance problem :

(one of the most applied concepts in search engines, address decoding, etc.)

↓

Given 2 strings, s1 = "abcd"
s2 = "acd",

edit distance between these 2 strings is defined as the min. number of operations performed to make s2 equal to s1, where the operations are:

(i) add a character
(ii) delete a character
(iii) replace a character

} at any index

↓

s1: anshuman s2: antihuman

delete t, and convert i → h } edit distance = 2

s1 : antihuman
s2 : anshuman

↓

single dimensional string to create multi-state recurrence

if (i or j reach s1, s2 lengths, return the opposite's remaining length)

edit-distance (s1, s2, i, j) // Base conditions

match from the beginning

check if they are same or not and on that basis decide how to move forward

↓

if same, just call on it+1, j+1,
else call on 3 operations and return (min of them + 1);

```
int same1 = edit_distance (s1, s2, i+1, j+1);
int same2 = edit_distance (s1, s2, i, j+1);
int same3 = edit_distance (s1, s2, i+1, j);
return 1+min (same1, min (same2, same3));
```

abcd acd a b c d

i) insert (i, j+1)
ii) replace (i+1, j+1)
iii) delete (i+1, j)
} 3 subproblems.

→ memoize for i & j
↓
build dp table in the opposite direction.
↓
thinking of the base case is important
↓
(n-1, m-1)

* if we start from behind, then building dp table will start from 0,0, and will be easier to code.

* time complexity : O(N*M)
↳ number of calls when the function code is executed.
↓
before memoization O(3^n)

* space complexity : O(N*M)

at any point of time, we are only looking at either the prefixs of both strings or suffixes of both strings. Hence we can replace the arguments or dependables as

* each operation has different cost ?
add the cost with each call & take their min., rather than adding 1 to min of all.

* google : There are two files, we need to maximise matching lines.
↓
treating a line as a fundamental block and comparing them
↓
maximising matching lines
↑
minimise operations

* for swap operation, we check for the two characters and if they match we call then on i+2, i+2, else we perform the other 3.