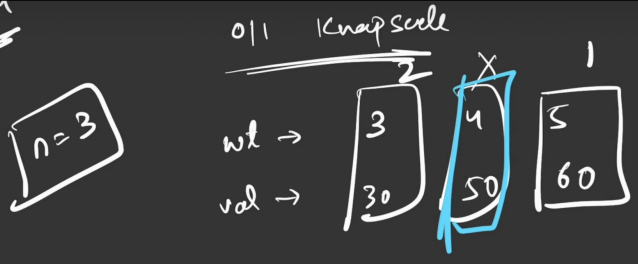
loc20 0/1 knapsack | recursion to single array space optimised approach

Recursion

try out all combinations
↓
take the best total value combination

DP-14

0/1 knapsack



① express everything in terms of index & write base cases

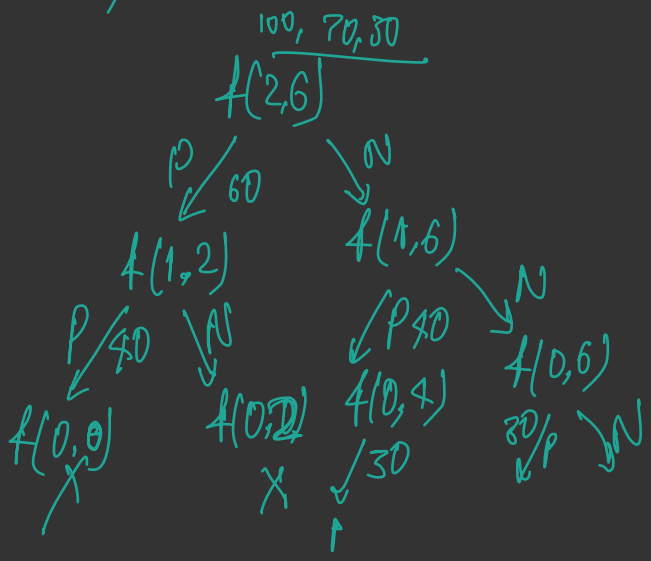
(ind, w)
↑ bag weight

② explore all possibilities
pick & not pick

③ max (all possibilities)

wt → 3, 2, 4
val → 30 40 60
ind → 0 1 2

f(n-1, 6)




```
f(ind, w)
{ if(ind == 0)
  { if (wt[0] ≤ w) return val[0];
    else return 0;
  }
  notTake = 0 + f(ind-1, w)
  take = INT_MAX
  if(wt[ind] ≤ w) pick = val[ind] + f(ind-1, w-wt[ind])
  return max(pick, notpick)
}
```

TC → 2^N

SC → O(N)

memoization

changing params → ind, W

ind: 0 → N-1

W: 0 → W

dp [N] [W+1]

TC → O(N x W)

SC → O(N x W) + O(N)

ASS

Tabulation

① base cases

② changing params in loops

③ copy the recurrence

dp[N][W+1] = 0

ind → 1 to n-1

wt → 0 to W

```
function f(ind, wt, val, W) {
    if(ind == 0) {
        if(wt[ind] <= W) return val[ind];
        else return 0;
    }

    let notTake = 0 + f(ind-1, wt, val, W);
    let take = Number.MIN_SAFE_INTEGER;
    if(wt[ind] <= W) take = val[ind] + f(ind-1, wt, val, W - wt[ind]);

    return Math.max(notTake, take);
}

function knapsack(wt, val, n, W) {
    return f(n-1, wt, val, W);
}

const value = [5,4,8,6];
const weight = [1,2,4,5];

knapsack(weight, value,4,5); //? 13
```

Recursion

```
function f(ind, wt, val, W, dp) {
    if(ind == 0) {
        if(wt[ind] <= W) return val[ind];
        else return 0;
    }
    if(dp[ind][W] != -1) return dp[ind][W];

    let notTake = 0 + f(ind-1, wt, val, W, dp);
    let take = Number.MIN_SAFE_INTEGER;
    if(wt[ind] <= W) take = val[ind] + f(ind-1, wt, val, W - wt[ind], dp);

    return dp[ind][W] = Math.max(notTake, take);
}

function knapsack(wt, val, n, W) {
    let dp = Array(n).fill().map(() => Array(W+1).fill(-1));
    return f(n-1, wt, val, W, dp);
}

const value = [5,4,8,6];
const weight = [1,2,4,5];

knapsack(weight, value,4,5); //? 13
```

memoization

```
function knapsack(wt, val, n, W) {
    let dp = Array(n)
        .fill()
        .map(() => Array(W + 1).fill(0));

    for (let i = wt[0]; i < W; i++) dp[0][i] = val[0];

    for (let ind = 1; ind < n; ind++) {
        for (let w = 0; w <= W; w++) {
            let notTake = 0 + dp[ind - 1][w];
            let take = Number.MIN_SAFE_INTEGER;
            if (wt[ind] <= w) take = val[ind] + dp[ind - 1][w - wt[ind]];
            dp[ind][w] = Math.max(notTake, take);
        }
    }

    return dp[n - 1][W];
}
```

Tabulation

```
function knapsack(wt, val, n, W) {

    let prev = Array(W+1).fill(0);
    let cur = Array(W+1).fill(0);

    for (let i = wt[0]; i < W; i++) prev[i] = val[0];

    for (let ind = 1; ind < n; ind++) {
        for (let w = 0; w <= W; w++) {
            let notTake = 0 + prev[w];
            let take = Number.MIN_SAFE_INTEGER;
            if (wt[ind] <= w) take = val[ind] + prev[w - wt[ind]];
            cur[w] = Math.max(notTake, take);
        }
        prev = [...cur]
    }

    return prev[W];
}
```

Space optimization

```js
function knapsack(wt, val, n, W) {
    let prev = Array(W+1).fill(0);

    for (let i = wt[0]; i < W; i++) prev[i] = val[0];

    for (let ind = 1; ind < n; ind++) {
        for (let w = 0; w <= W; w++) {
            let notTake = 0 + prev[w];
            let take = Number.MIN_SAFE_INTEGER;
            if (wt[ind] <= w) take = val[ind] + prev[w - wt[ind]];
            prev[w] = Math.max(notTake, take);
        }
    }

    return prev[W];
}
```

1 row optimization

---

lec21 minimum coins. infinite supplies [coins]

trying out all combos to find target

take the combo that has min coins

knapsack.

infinite supply or multiple supply

pick → ind will remain same

target will update

Problem Statement Suggest Edit

You are given an array of 'N' distinct integers and an integer 'X' representing the target sum. You have to tell the minimum number of elements you have to take to reach the target sum 'X'.

Note:

You have an infinite number of elements of each type.

For Example

```
If N=3 and X=7 and array elements are [1,2,3].
Way 1 - You can take 4 elements  [2, 2, 2, 1] as 2 + 2 + 2 + 1 = 7.
Way 2 - You can take 3 elements  [3, 3, 1] as 3 + 3 + 1 = 7.
Here, you can see in Way 2 we have used 3 coins to reach the target sum of 7.
Hence the output is 3.
```

Recussion

TC → >> $O(2^N)$ exponential

SC → >> $O(N)$

```
f(ind, target)
{ if (ind == 0)
    { if ((target % num[ind]) == 0) return target/num[ind]
      return MAX_INT;
    }
  notpick = 0 + f(ind-1, target)
  pick = INT_MAX
  if( num[ind] <= target) pick = 1+f(ind, target - num[ind])
  return Math.min(pick, notPick)
}
```

memoization

ind, target

dp[N][target+1]

TC → O(N x Target)

SC → O(N x Target) + O(Target)
ASS

Tabulation

① base case

② changing params ind, T

③ copy recurrence

dp[N][target+1]

for (T=0 → target)

if (T % a[0] == 0)

dp[0][T] = T/a[0]

else

dp[0][T] = INT_MAX

```
function f(ind, T, num) {
    if (ind == 0) {
        if ((T % num[ind]) == 0) return T/num[ind];
        else return Number.MAX_SAFE_INTEGER;
    }

    let notTake = 0 + f(ind - 1, T, num);
    let take = Number.MAX_SAFE_INTEGER;
    if (num[ind] <= T) take = 1 + f(ind, T - num[ind], num);

    return Math.min(notTake, take);
}

function minimumElements(T, num) {
    let n = num.length;
    return f(n-1, T, num);
}

const num = [1, 2, 3];
minimumElements(7, num); //? 3
```

Recursion

```
function f(ind, T, num, dp) {
    if (ind == 0) {
        if ((T % num[ind]) == 0) return T/num[ind];
        else return Number.MAX_SAFE_INTEGER;
    }
    if(dp[ind][T] != -1) return dp[ind][T];
    let notTake = 0 + f(ind - 1, T, num, dp);
    let take = Number.MAX_SAFE_INTEGER;
    if (num[ind] <= T) take = 1 + f(ind, T - num[ind], num, dp);

    return dp[ind][T] = Math.min(notTake, take);
}

function minimumElements(T, num) {
    let n = num.length;
    let dp = Array(n).fill().map(() => Array(T+1).fill(-1));
    return f(n-1, T, num, dp);
}

const num = [1, 2, 3];
minimumElements(7, num); //? 3
```

memoization

```javascript
function minimumElements(T, num) {
    let n = num.length;
    let dp = Array(n)
        .fill()
        .map(() => Array(T + 1).fill(0));

    for (let t = 0; t <= T; t++) {
        if (t % num[0] == 0) {
            dp[0][t] = t / num[0];
        } else {
            dp[0][t] = Number.MAX_SAFE_INTEGER;
        }
    }

    for (let ind = 1; ind < n; ind++) {
        for (let t = 0; t <= T; t++) {
            let notTake = 0 + dp[ind - 1][t];
            let take = Number.MAX_SAFE_INTEGER;
            if (num[ind] <= t) take = 1 + dp[ind][t - num[ind]];

            dp[ind][t] = Math.min(notTake, take);
        }
    }

    return dp[n - 1][T];
}

const num = [1, 2, 3];
minimumElements(7, num); //? 3
```

Tabulation

```javascript
function minimumElements(T, num) {
    let n = num.length;
    let dp = Array(n)
        .fill()
        .map(() => Array(T + 1).fill(0));

    let prev = Array(T+1).fill(0);
    let cur = Array(T+1).fill(0);

    for (let t = 0; t <= T; t++) {
        if (t % num[0] == 0) {
            prev[t] = t / num[0];
        } else {
            prev[t] = Number.MAX_SAFE_INTEGER;
        }
    }

    for (let ind = 1; ind < n; ind++) {
        for (let t = 0; t <= T; t++) {
            let notTake = 0 + prev[t];
            let take = Number.MAX_SAFE_INTEGER;
            if (num[ind] <= t) take = 1 + cur[t - num[ind]];

            cur[t] = Math.min(notTake, take);
        }
        prev = [...cur]
    }

    return prev[T];
}

const num = [1, 2, 3];
minimumElements(7, num); //? 3
```

space optimization

---

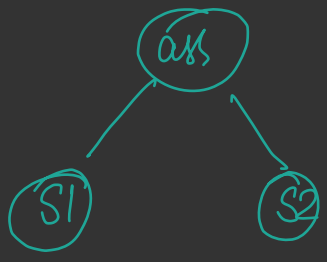
lec22 target Sum subsequence

arr = [1, 2, 3, 1]

target = 3

1st way → - 1 + 2 + 3 - 1

2nd way → + 1 - 2 + 3 + 1



(2,3) − (1,1) ⇒ 3

(1,3,1) − (2) ⇒ 3

S1 − S2 = target

count (S1 - S2) : Difference

↳ lec18.

Problem Statement Suggest Edit

You are given an array 'ARR' of 'N' integers and a target number, 'TARGET'. Your task is to build an expression out of an array by adding one of the symbols '+' and '-' before each integer in an array, and then by concatenating all the integers, you want to achieve a target. You have to return the number of ways the target can be achieved.

For Example :

```
You are given the array 'ARR' = [1, 1, 1, 1, 1], 'TARGET' =
3. The number of ways this target can be achieved is:
1. -1 + 1 + 1 + 1 + 1 = 3
2. +1 - 1 + 1 + 1 + 1 = 3
3. +1 + 1 - 1 + 1 + 1 = 3
4. +1 + 1 + 1 - 1 + 1 = 3
5. +1 + 1 + 1 + 1 - 1 = 3
These are the 5 ways to make. Hence the answer is 5.
```

```javascript
function findWays(target, arr) {
    let n = arr.length;
    let prev = Array(target+1).fill(0);
    let cur = Array(target+1).fill(0);

    if(arr[0] == 0) prev[0] = 2;
    else prev[0] = 1;

    for(let i=0; i<n; i++) prev[0] = 1;
    if(arr[0] != 0 && arr[0] <= target) prev[arr[0]] = 1;

    for(let ind=1; ind<n; ind++) {
        for(let sum=0; sum<=target; sum++) {
            let notPick = prev[sum];
            let pick = 0;
            if(arr[ind] <= sum) pick = prev[sum - arr[ind]];

            cur[sum] = notPick + pick;
        }
        prev = [...cur];
    }

    return prev[target];
}

function countPartitions(n, d, arr) {
    let totalSum = 0;
    for(let item of arr) totalSum += item;

    if(totalSum - d < 0 || (totalSum-d)%2) return 0;
    return findWays((totalSum-d)/2, arr);
}

function targetSum(n, d, arr) {
    return countPartitions(n, d, arr);
}

const num = [1, 2, 3, 1];
targetSum(4, 3, num); //?
```

space optimized

loc 23. count change 2, infinite supply problems

Ways To Make Coin Change

Contributed by **Ankit Kharb**
Last Updated: 23 Feb, 2023

Medium | 0/80 | Avg time to solve 20 mins | Success Rate 80 %

Share | 67 upvotes

Problem Statement Suggest Edit

You are given an infinite supply of coins of each of denominations D = {D0, D1, D2, D3, ... Dn-1}. You need to figure out the total number of ways W, in which you can make a change for value V using coins of denominations from D. Print 0, if a change isn't possible.

Detailed explanation (Input/output format, Notes, Images)

Sample Input 1 :

```
3
1 2 3
4
```

Sample Output 1:

```
4
```

Explanation For Sample Output 1:

```
We can make a change for the value V = 4 in four ways.
1. (1,1,1,1),
2. (1,1, 2), [One thing to note here is, (1, 1, 2) is same as
that of (2, 1, 1) and (1, 2, 1)]
3. (1, 3), and
4. (2, 2)
```

Recursion

① express in (ind, target)
② base cases
③ explore all possibilities
④ sum all ways & return

TC → exponential
SC → O(target)

```
f(ind, T)
{  if (ind==0){
     if( T % arr[ind] ==0) return 1 else return 0;
   }
   notTake = f(ind-1, T)
   take = 0
   if( arr[ind] ≤ T) take = f(ind, T-a[ind])
   return take + notTake
}
```

Memoization

changing params → ind, Target
ind: 0 to n-1
Target: 0 to T
dp[N][T+1]

TC → O(N×T)
SC → O(N×T) + O(target) Ass

Tabulation

```
dp[N][T+1]
for( T=0 to target)
    dp[0][T] = ( T % arr[0] ==0) ? 1 : 0;
for( ind=1; ind<n; ind++)
{   for (T=0; T≤target; T++)
    {   notTake = dp[ind-1][T]
        take = 0
        if( a[ind] <=T) take = dp[ind][T-a[ind]]
```

dp[ind][T] = take + notTake

}

}

return dp[n-1][value]

```
function f(ind, T, arr) {
    if(ind == 0) {
        return T % arr[ind] === 0 ? 1 : 0;
    }

    let notTake = f(ind-1, T, arr);
    let take = 0;
    if(arr[ind] <= T) take = f(ind, T - arr[ind], arr);

    return take + notTake;
}

function countWaysToMakeChange(n, target, arr) {
    return f(n-1, target, arr);
}

const num = [1, 2, 3];
countWaysToMakeChange(3, 4, num); //?
```

```
function f(ind, T, arr, dp) {
    if(ind == 0) {
        return T % arr[ind] === 0 ? 1 : 0;
    }
    if(dp[ind][T] != -1) return dp[ind][T];

    let notTake = f(ind-1, T, arr, dp);
    let take = 0;
    if(arr[ind] <= T) take = f(ind, T - arr[ind], arr, dp);

    return dp[ind][T] = take + notTake;
}

function countWaysToMakeChange(n, target, arr) {
    let dp = Array(n).fill().map(() => Array(target+1).fill(-1));
    return f(n-1, target, arr, dp);
}

const num = [1, 2, 3];
countWaysToMakeChange(3, 4, num); //?
```

```
function countWaysToMakeChange(n, target, arr) {
    let dp = Array(n).fill().map(() => Array(target+1).fill(0));

    for(let T=0; T<=target; T++) dp[0][T] = (T % arr[0] === 0) ? 1 : 0;

    for(let ind=1; ind<n; ind++) {
        for(let T=0; T<=target; T++) {
        let notTake = dp[ind - 1][T];
        let take = 0;
        if (arr[ind] <= T) take = dp[ind][T - arr[ind]];
        dp[ind][T] = take + notTake;
        }
    }

    return dp[n-1][target];
}

const num = [1, 2, 3];
countWaysToMakeChange(3, 4, num); //?
```

```
function countWaysToMakeChange(n, target, arr) {
    let dp = Array(n).fill().map(() => Array(target+1).fill(0));

    let prev = Array(target+1).fill(0);
    let cur = Array(target+1).fill(0);

    for(let T=0; T<=target; T++) prev[T] = (T % arr[0] === 0) ? 1 : 0;

    for(let ind=1; ind<n; ind++) {
        for(let T=0; T<=target; T++) {
        let notTake = prev[T];
        let take = 0;
        if (arr[ind] <= T) take = cur[T - arr[ind]];
        cur[T] = take + notTake;
        }
        prev = [...cur];
    }

    return prev[target];
}

const num = [1, 2, 3];
countWaysToMakeChange(3, 4, num); //?
```

loc 24 unbounded knapsack

Unbounded Knapsack

Contributed by **Ambuj verma**
Last Updated: 23 Feb, 2023

Medium | 0/80 | Avg time to solve 15 mins | Success Rate 85 %

Share | 58 upvotes

Problem Statement

Suggest Edit

You are given 'N' items with certain 'PROFIT' and 'WEIGHT' and a knapsack with weight capacity 'W'. You need to fill the knapsack with the items in such a way that you get the maximum profit. You are allowed to take one item multiple times.

For Example

```
Let us say we have 'N' = 3 items and a knapsack of capacity
'W' =  10
'PROFIT' = { 5, 11, 13 }
'WEIGHT' = { 2, 4, 6 }

We can fill the knapsack as:

1 item of weight 6 and 1 item of weight 4.
1 item of weight 6 and 2 items of weight 2.
2 items of weight 4 and 1 item of weight 2.
5 items of weight 2.

The maximum profit will be from case 3 i.e '27'. Therefore
maximum profit = 27.
```

Constraints

```
1 <= T <= 50
1 <= N <= 10^3
1 <= W <= 10^3
1 <= PROFIT[ i ] , WEIGHT[ i ] <= 10^8

Time Limit: 1sec
```

Recursion

f(ind, w)
{ if(ind == 0) {
return $\left(\frac{w}{wt[0]}\right) \times val[0]$
}
notTake = 0 + f(ind-1, w)
take = INT_MIN
if(wt[ind] ≤ w) take = val[ind] + f(ind, w − wt[ind])
return max(take, notTake)
}

TC → exponential
(not 2^N because we are standing at same index)

SC → O(wt)

memoization

ind, wt
dp[N][wt+1]

Tabulation

① Base case
② loops. ind → 1 to n-1
w → 0 to wt
③ copy the recurrence

```
function f(ind, W, profit, weight) {
    if(ind == 0) {
        return ((W/weight[ind]) * profit[ind])
    }

    let notTake = 0 + f(ind-1, W, profit, weight);
    let take = Number.MIN_SAFE_INTEGER;
    if(weight[ind] <= W) take = profit[ind] + f(ind, W - weight[ind], profit, weight);

    return Math.max(take, notTake);
}

function unboundedKnapsack(W, profit, weight) {
    let n = profit.length;
    return f(n-1, W, profit, weight);
}

const profit = [5, 11, 13];
const weight = [2, 4, 6];
unboundedKnapsack(10, profit, weight); //? 27
```

recursion

```
function f(ind, W, profit, weight, dp) {
    if(ind == 0) {
        return ((W/weight[ind]) * profit[ind])
    }

    if(dp[ind][W] != -1) return dp[ind][W];

    let notTake = 0 + f(ind-1, W, profit, weight, dp);
    let take = Number.MIN_SAFE_INTEGER;
    if(weight[ind] <= W) take = profit[ind] + f(ind, W - weight[ind], profit, weight, dp);

    return dp[ind][W] = Math.max(take, notTake);
}

function unboundedKnapsack(W, profit, weight) {
    let n = profit.length;
    let dp = Array(n).fill().map(() => Array(W+1).fill(-1));
    return f(n-1, W, profit, weight, dp);
}

const profit = [5, 11, 13];
const weight = [2, 4, 6];
unboundedKnapsack(10, profit, weight); //? 27
```

memoization

```javascript
function unboundedKnapsack(W, profit, weight) {
    let n = profit.length;
    let dp = Array(n).fill().map(() => Array(W+1).fill(0));

    for(let w=0; w<=W; w++) dp[0][w] = (w / weight[0]) * profit[0];

    for(let ind=1; ind<n; ind++) {
        for(let w=0; w<= W; w++) {
            let notTake = 0 + dp[ind - 1][w];
            let take = Number.MIN_SAFE_INTEGER;
            if (weight[ind] <= w)
                take =
                    profit[ind] + dp[ind][w - weight[ind]];

            dp[ind][w] = Math.max(take, notTake);
        }
    }

    return dp[n-1][W];
}

const profit = [5, 11, 13];
const weight = [2, 4, 6];
unboundedKnapsack(10, profit, weight); //? 27
```

tabulation

Todo -> optimize it to 1 array

```javascript
function unboundedKnapsack(W, profit, weight) {
    let n = profit.length;
    let prev = Array(W+1).fill(0);
    let cur = Array(W+1).fill(0);

    for(let w=0; w<=W; w++) prev[w] = (w / weight[0]) * profit[0];

    for(let ind=1; ind<n; ind++) {
        for(let w=0; w<= W; w++) {
            let notTake = 0 + prev[w];
            let take = Number.MIN_SAFE_INTEGER;
            if (weight[ind] <= w)
                take =
                    profit[ind] + cur[w - weight[ind]];

            cur[w] = Math.max(take, notTake);
        }
        prev = [...cur]
    }

    return prev[W];
}

const profit = [5, 11, 13];
const weight = [2, 4, 6];
unboundedKnapsack(10, profit, weight); //? 27
```

space optimization

```javascript
function unboundedKnapsack(W, profit, weight) {
    let n = profit.length;
    let prev = Array(W+1).fill(0);

    for(let w=0; w<=W; w++) prev[w] = (w / weight[0]) * profit[0];

    for(let ind=1; ind<n; ind++) {
        for(let w=0; w<= W; w++) {
            let notTake = 0 + prev[w];
            let take = Number.MIN_SAFE_INTEGER;
            if (weight[ind] <= w)
                take =
                    profit[ind] + prev[w - weight[ind]];

            prev[w] = Math.max(take, notTake);
        }
    }

    return prev[W];
}

const profit = [5, 11, 13];
const weight = [2, 4, 6];
unboundedKnapsack(10, profit, weight); //?
```

lec25 -> Rod cutting problem, 1D array space optimised

Rod cutting problem

Contributed by **Mutiur Rehman khan**
Last Updated: 23 Feb, 2023

Medium | 0/80 | Avg time to solve 40 mins | Success Rate 75 %

Share | 125 upvotes

Problem Statement Suggest Edit

Given a rod of length 'N' units. The rod can be cut into different sizes and each size has a cost associated with it. Determine the maximum cost obtained by cutting the rod and selling its pieces.

Note:

1. The sizes will range from 1 to 'N' and will be integers.
2. The sum of the pieces cut should be equal to 'N'.
3. Consider 1-based indexing.

Sample Input 1:

```
2
5
2 5 7 8 10
8
3 5 8 9 10 17 17 20
```

Sample Output 1:

```
12
24
```

Explanation Of Sample Input 1:

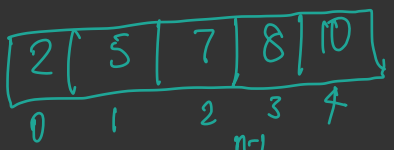
```
Test case 1:

All possible partitions are:
1,1,1,1,1           max_cost=(2+2+2+2+2)=10
1,1,1,2             max_cost=(2+2+2+5)=11
1,1,3               max_cost=(2+2+7)=11
1,4                 max_cost=(2+8)=10
5                   max_cost=(10)=10
2,3                 max_cost=(5+7)=12
1,2,2               max _cost=(1+5+5)=12

Clearly, if we cut the rod into lengths 1,2,2, or 2,3, we get
the maximum cost which is 12.
```

Recursion

① explore in → (ind, N)

② explore all possibilities → not take / take

③ return max of all possibilities



$f(4, N)$

till index 4, what is the max price obtained?

```
f(ind, N){
    if(ind==0){
        return (N x price[0]);
    }
    notTake = 0 + f(ind-1, N)
    take = INT_MIN
    rodLength = ind+1
    if(rodLength <= N) take = price[ind] + f(ind, N-rodLength)
    return max(take, notTake)
}
```

TC → exponential (greater than 2^N)

SC → O(target)

Memoization

ind, N

dp[N][N+1]

TC → O(N x N)

SC → O(N x N) + O(target)
ASS

Tabulation

① write base case

② changing params

③ recurrence

dp[N][N+1]

for(N => 0 to N) dp[0][N] = N x price[0]

for(ind => 1 to N-1)

for(N => 0 to N)

recurrence

```
function f(ind, N, price) {
    if(ind == 0) {
        return N * price[ind];
    }

    let notTake = 0 + f(ind-1, N, price);
    let take = Number.MIN_SAFE_INTEGER;
    let rodLength = ind+1;
    if(rodLength <= N) take = price[ind] + f(ind, N-rodLength, price);

    return Math.max(take, notTake);
}

function cutRod(N, price) {
    return f(N-1, N, price);
}

const price = [2,5,7,8,10];
cutRod(5, price); //? 12
```

recursion

```
function f(ind, N, price, dp) {
    if(ind == 0) {
        return N * price[ind];
    }

    if(dp[ind][N] != -1) return dp[ind][N];

    let notTake = 0 + f(ind-1, N, price, dp);
    let take = Number.MIN_SAFE_INTEGER;
    let rodLength = ind+1;
    if(rodLength <= N) take = price[ind] + f(ind, N-rodLength, price, dp);

    return dp[ind][N] = Math.max(take, notTake);
}

function cutRod(N, price) {
    let dp = Array(N).fill().map(() => Array(N+1).fill(-1));
    return f(N-1, N, price, dp);
}

const price = [2,5,7,8,10];
cutRod(5, price); //? 12
```

memorization

```
function cutRod(N, price) {
    let dp = Array(N).fill().map(() => Array(N+1).fill(0));

    for(let n=0; n<=N; n++) dp[0][n] = n * price[0];

    for(let ind=1; ind<N; ind++) {
        for(let n=0; n<=N; n++) {
            let notTake = 0 + dp[ind - 1][n];
            let take = Number.MIN_SAFE_INTEGER;
            let rodLength = ind + 1;
            if (rodLength <= n)
                take = price[ind] + dp[ind][n - rodLength];

            (dp[ind][n] = Math.max(take, notTake));
        }
    }

    return dp[N-1][N];
}

const price = [2,5,7,8,10];
cutRod(5, price); //? 12
```

Tabulation

```
function cutRod(N, price) {
    let prev = Array(N+1).fill(0);
    let cur = Array(N+1).fill(0);

    for(let n=0; n<=N; n++) prev[n] = n * price[0];

    for(let ind=1; ind<N; ind++) {
        for(let n=0; n<=N; n++) {
            let notTake = 0 + prev[n];
            let take = Number.MIN_SAFE_INTEGER;
            let rodLength = ind + 1;
            if (rodLength <= n)
                take = price[ind] + cur[n - rodLength];

            (cur[n] = Math.max(take, notTake));
        }
        prev = [...cur]
    }

    return prev[N];
}

const price = [2,5,7,8,10];
cutRod(5, price); //? 12
```

space optimization

```
function cutRod(N, price) {
    let prev = Array(N+1).fill(0);

    for(let n=0; n<=N; n++) prev[n] = n * price[0];

    for(let ind=1; ind<N; ind++) {
        for(let n=0; n<=N; n++) {
            let notTake = 0 + prev[n];
            let take = Number.MIN_SAFE_INTEGER;
            let rodLength = ind + 1;
            if (rodLength <= n)
                take = price[ind] + prev[n - rodLength];

            (prev[n] = Math.max(take, notTake));
        }
    }

    return prev[N];
}

const price = [2,5,7,8,10];
cutRod(5, price); //? 12
```

1D optimized

lec26 longest common subsequences | DP on strings

Longest Common Subsequence

Contributed by **Deep Mavani**
Last Updated: 23 Feb, 2023

Medium | 0/80 | Avg time to solve 39 mins

Share | 98 upvotes

Problem Statement Suggest Edit

Given two strings, 'S' and 'T' with lengths 'M' and 'N', find the length of the 'Longest Common Subsequence'.
For a string 'str'(per se) of length K, the subsequences are the strings containing characters in the same relative order as they are present in 'str,' but not necessarily contiguous. Subsequences contain all the strings of length varying from 0 to K.

Example :

```
Subsequences of string "abc" are:  ""(empty string), a, b, c,
ab, bc, ac, abc.
```

Detailed explanation (Input/output format, Notes, Images)

Constraints :

```
0 <= M <= 10 ^ 3
0 <= N <= 10 ^ 3

Time Limit: 1 sec
```

Sample Input 1 :

```
adebc
dcadb
```

Sample Output 1 :

```
3
```

Explanation Of The Sample Output 1 :

```
Both the strings contain a common subsequence 'adb', which is
the longest common subsequence with length 3.
```

s1 = 'adebc'
s2 = "dcadb"

DP on strings → comparison
→ replaces/edit

M1 Brute force → exponential

M2 → recursion
generate all subsequences & compare on the way

ex acd / ced

a	c
c	e
d	d
ac	ce
ad	cd
cd	ed
acd	ced

② subsequence

rules to write recursion
① express in index (ind1, ind2)
② explore all possibilities on that index
③ take best among them

f(2,2) → los of str1(0...2) & str2(0...2)

f(2,5) → str1(0...2) & str2(0...5)

recursion
① if match on indexes
1 + f(ind1-1, ind2-1)

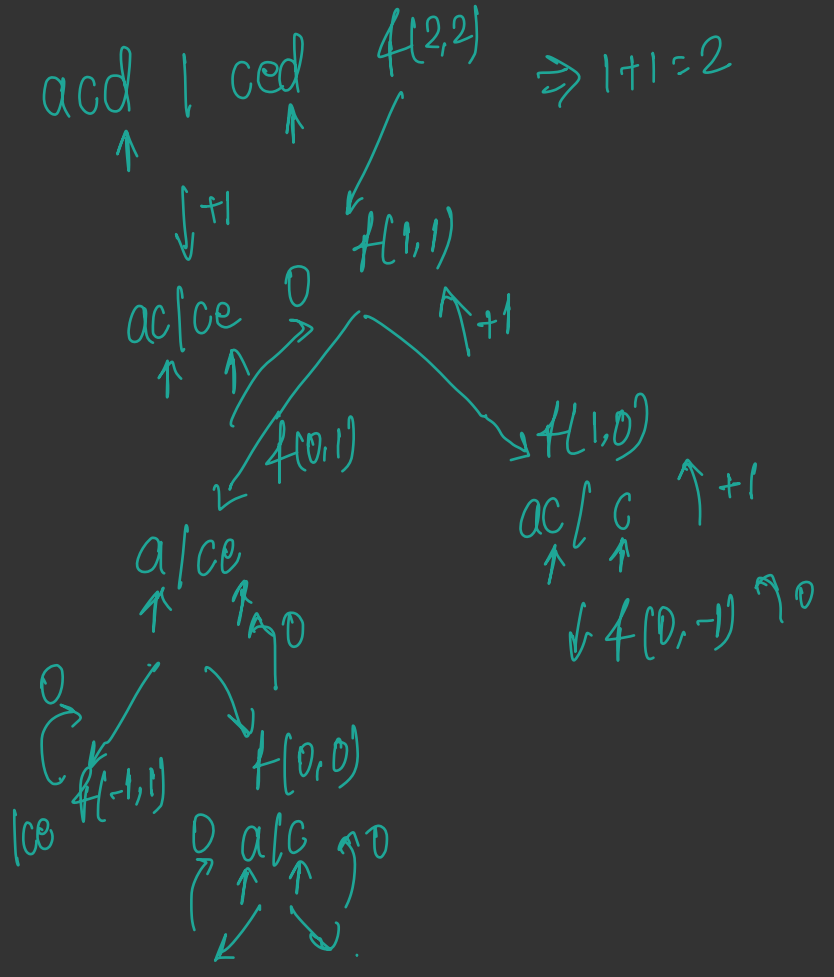
② not match

$0 + \max(f(ind1-1, ind2), f(ind1, ind2-1))$

TC → $2^n \times 2^m$ exponential

SC →

```
f(ind1, ind2)
{ if(ind1<0 || ind2<0)
      return 0;
  if(s1[ind1] == s2[ind2])
      return 1+f(ind1-1, ind2-1)
  return 0+max(f(ind1-1, ind2), f(ind1, ind2-1))
}
```



Memoization

$f(i,j)$ → lcs of $s1[0 \ldots i]$ & $s2[0 \ldots j]$

$dp[N][M] = -1$

TC → $O(N \times M)$

SC → $O(N \times M) + O(N+M)$ ASS

```javascript
function f(i, j, s1, s2) {
    if(i < 0 || j < 0) {
        return 0
    }

    if(s1[i] == s2[j]) {
        return 1 + f(i-1, j-1, s1, s2);
    }

    return Math.max(f(i-1, j, s1, s2), f(i, j-1, s1, s2));
}

function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    return f(n-1,m-1,s1,s2);
}

const s1 = "adebc";
const s2 = "dcadb";
lcs(s1, s2); //? 3
```

recursion

```javascript
function f(i, j, s1, s2, dp) {
    if(i < 0 || j < 0) {
        return 0
    }

    if(dp[i][j] != -1) return dp[i][j];

    if(s1[i] == s2[j]) {
        return dp[i][j] = 1 + f(i-1, j-1, s1, s2, dp);
    }

    return dp[i][j] = Math.max(f(i-1, j, s1, s2, dp), f(i, j-1, s1, s2, dp));
}

function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n).fill().map(() => Array(m).fill(-1));
    return f(n-1,m-1,s1,s2,dp);
}

const s1 = "adebc";
const s2 = "dcadb";
lcs(s1, s2); //? 3
```

memoization

Tabulation / bottom up

1. declare dp array
2. copy the base case
3. nested loops for changing parameters
4. copy the recurrence

```
dp[N+1][M+1] = 0

for(i = 0 to N) dp[i][0] = 0
for(j = 0 to M) dp[0][j] = 0

for(i = 1 to N)
    for(j = 1 to M)
    {
    :
    :
```

1. dp[N][M] = 0
2. base cases are written for → but in array we can't go to negative index

shifting of index

```
-1  0  1  2  3 ... n-1
 ↑  ↑                ↑
f(0) f(1)          f(n,m)
```

shifting it to right by 1

so the base case

if(i<0 || j<0) return 0

will become

if(i==0 || j==0) return 0

```javascript
function f(i, j, s1, s2, dp) {
    if(i == 0 || j == 0) {
        return 0
    }

    if(dp[i][j] != -1) return dp[i][j];

    if(s1[i-1] == s2[j-1]) {
        return dp[i][j] = 1 + f(i-1, j-1, s1, s2, dp);
    }

    return dp[i][j] = Math.max(f(i-1, j, s1, s2, dp), f(i, j-1, s1, s2, dp));
}

function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n+1).fill().map(() => Array(m+1).fill(-1));
    return f(n,m,s1,s2,dp);
}

const s1 = "adebc";
const s2 = "dcadb";
lcs(s1, s2); //? 3
```

recursion

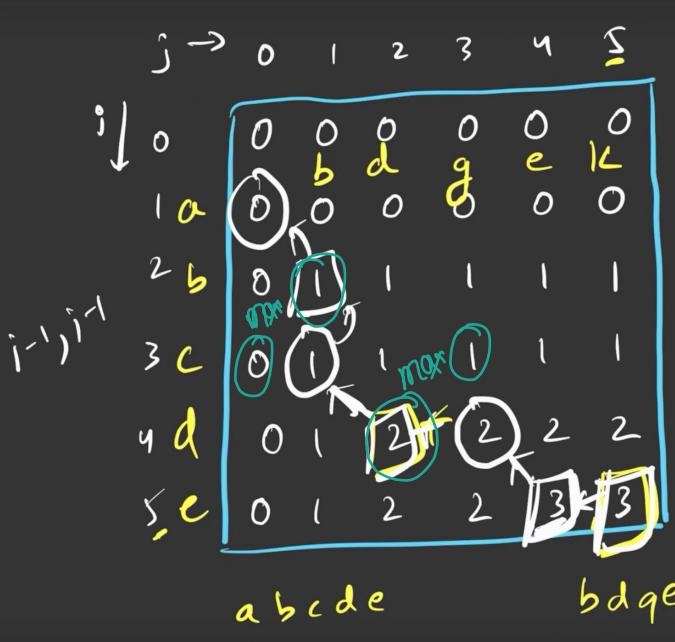
```js
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n + 1)
        .fill()
        .map(() => Array(m + 1).fill(-1));

    for (let i = 0; i <= n; i++) dp[i][0] = 0;
    for (let j = 0; j <= m; j++) dp[0][j] = 0;

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = 1 + dp[i-1][j-1];
            } else {
                dp[i][j] = Math.max(dp[i-1][j], dp[i][j]);
            }
        }
    }

    return dp[n][m];
}

const s1 = "adebc";
const s2 = "dcadb";
lcs(s1, s2); //? 3
```

Tabulation

```js
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(0);
    let cur = Array(n+1).fill(0);

    for (let j = 0; j <= m; j++) prev[j] = 0;

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                cur[j] = 1 + prev[j-1];
            } else {
                cur[j] = Math.max(prev[j], cur[j]);
            }
        }
        prev = [...cur];
    }

    return prev[m];
}

const s1 = "adebc";
const s2 = "dcadb";
lcs(s1, s2); //? 3
```

space optimization

lec27 → print longest common subsequence



b d e

len = dp[n][m], index = len-1

s = " " ; for (i=0 to len-1) s += ' '

i = n ; j = m

while (i > 0 && j > 0)

{ if (s1[i-1] == s2[j-1])

{

s[index] = s1[i-1];

index--

i--; j--;

}

else if (dp[i-1][j] > dp[i][j-1])

{ i = i-1;

}

else

{ j = j-1;

}

}

if s1[i-1] == s2[j-1]

dp[i][j] = 1 + dp[i-1][j-1]

else

dp[i][j] = max(dp[i-1][j], dp[i][j-1])

```javascript
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n + 1)
        .fill()
        .map(() => Array(m + 1).fill(-1));

    for (let j = 0; j <= m; j++) dp[0][j] = 0;
    for (let i = 0; i <= n; i++) dp[i][0] = 0
    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = 1 + dp[i-1][j-1];
            } else {
                dp[i][j] = Math.max(dp[i-1][j], dp[i][j]);
            }
        }
    }

    let len = dp[n][m];
    let index = len-1;
    let s = [];
    for(let i=0; i<len; i++) s[i]= '$';

    let i=n; let j=m;
    while(i>0 && j>0) {
        if(s1[i-1] == s2[j-1]) {
            s[index] = s1[i-1];
            index--;
            i--;
            j--;
        }
        else if(dp[i-1][j] > dp[i][j-1]) {
            i = i-1;
        }
        else {
            j = j-1;
        }
    }
    return s.join('');
}

const s1 = "adebc";
const s2 = "dcadb";
lcs(s1, s2); //? adb
```

lec 28. longest common substring (not subsequence)

Longest Common Substring

Contributed by **Deep Mavani**
Last Updated: 23 Feb, 2023

Medium | 0/80 | Avg time to solve 25 mins | Success Rate 75 %

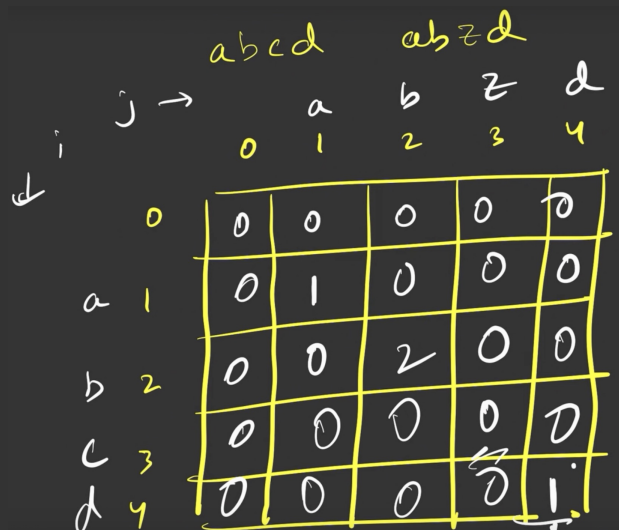
Share | 53 upvotes

Problem Statement Suggest Edit

You have been given two strings 'STR1' and 'STR2'. You have to find the length of the longest common substring.
A string “s1” is a substring of another string “s2” if “s2” contains the same characters as in “s1”, in the same order and in continuous fashion also.
For Example :

```
If 'STR1' = “abcjklp” and 'STR2' = “acjkp”  then the output will be 3.

Explanation: The longest common substring is “cjk” which is of length 3.
```

s1 = abcd
s2 = abzd



```js
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n + 1)
        .fill()
        .map(() => Array(m + 1).fill(-1));

    for (let j = 0; j <= m; j++) dp[0][j] = 0;
    for (let i = 0; i <= n; i++) dp[i][0] = 0;

    let ans = 0;

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = 1 + dp[i - 1][j - 1];
                ans = Math.max(ans, dp[i][j]);
            } else {
                dp[i][j] = 0;
            }
        }
    }

    return ans;
}

const s1 = "abcd";
const s2 = "abzd";
lcs(s1, s2); //? 2
```

tabulation

```js
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(0);
    let cur = Array(m+1).fill(0);

    let ans = 0;

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                cur[j] = 1 + prev[j - 1];
                ans = Math.max(ans, cur[j]);
            } else {
                cur[j] = 0;
            }
        }
        prev = [...cur];
    }

    return ans;
}

const s1 = "abcd";
const s2 = "abzd";
lcs(s1, s2); //? 2
```

space optimization

loc29 - longest palindromic subsequence

palindrome → if it reads same way forward & backward

palindrome

s = bbbab → ab ✗

bb ✓

bbb ✓

bbbb → 4

bab ✓

M1 → generate all subsequences, & check for palindrome & take the longest

M2 → s1 → given input

s2 → reverse s1

call lcs(s1, s2)

loc-26

```javascript
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n+1).fill().map(() => Array(m+1).fill(0));

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = 1 + dp[i-1][j-1];
            } else {
                dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }

    return dp[n][m];
}

function longestPalindrome(s1) {
    let s2 = s1.split('').reverse().join('');
    return lcs(s1, s2);
}

const s = "bbbab";
longestPalindrome(s); //?
```

Tabulation

```javascript
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(0);
    let cur = Array(m+1).fill(0);

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                cur[j] = 1 + prev[j-1];
            } else {
                cur[j] = Math.max(prev[j], cur[j-1]);
            }
        }
        prev = [...cur];
    }

    return prev[m];
}

function longestPalindrome(s1) {
    let s2 = s1.split('').reverse().join('');
    return lcs(s1, s2);
}

const s = "bbbab";
longestPalindrome(s); //?
```

space optimized

---

lec 30 => min insertion to make string palindrome

you can insert any char anywhere

s = abcaa

max operations = len(s)

abcaa aacba

reverse of i/p

s = abcaa

↓

aabcbaa (2) insertions

How do you approach?

keep the longest palindromic portion intact

ex-1 abcaa len => 5

longest palindrome length => 3

min insertion => 5-3 = 2

abcacba

ex-2

codingninja

length => 11

palindrome length => 11-5 = 6

codajningninjadoc

```javascript
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(0);
    let cur = Array(m+1).fill(0);

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                cur[j] = 1 + prev[j-1];
            } else {
                cur[j] = Math.max(prev[j], cur[j-1]);
            }
        }
        prev = [...cur];
    }

    return prev[m];
}

function longestPalindrome(s1) {
    let s2 = s1.split('').reverse().join('');
    return lcs(s1, s2);
}

function minInsertion(s) {
    let n = s.length;
    let palindromeLength = longestPalindrome(s);
    return n - palindromeLength;
}

const s = "codingninja";
minInsertion(s); //? 6
```

space optimized

---

lec31 -> min insertions/deletions to convert string A to string B

s1 = "abcd"
s2 = "anc"

convert s1 to s2

(M) ① delete everything from s1 -> 4 operations
② insert everything from s2 to s1 -> 3 "
total : 7 operations

max operations = n + m
↑ deletions ↑ insertions

a b c d
2 deletions
a c
a n c
1 insert
ac -> longest common subsequence

lcs = 2
n = 4
m = 3
ans => 4-2 + 3-2
= 7-4
= 3
=> n+m-lcs×2

```javascript
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n+1).fill().map(() => Array(m+1).fill(0));

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = 1 + dp[i-1][j-1];
            } else {
                dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }

    return dp[n][m];
}

function convert(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let len = lcs(s1,s2);
    return (m+n) - 2* len;
}

const s1 = "abcd";
const s2 = "anc";
convert(s1, s2); //?3
```

tabulation

Loc 32 → shortest common supersequence

S1 = "brute" S2 = "groot"

"brutegroot" length = 10

"bgruoote" length = 8 → shortest

S1 = bleed S2 = blue

ans → bleued

brute

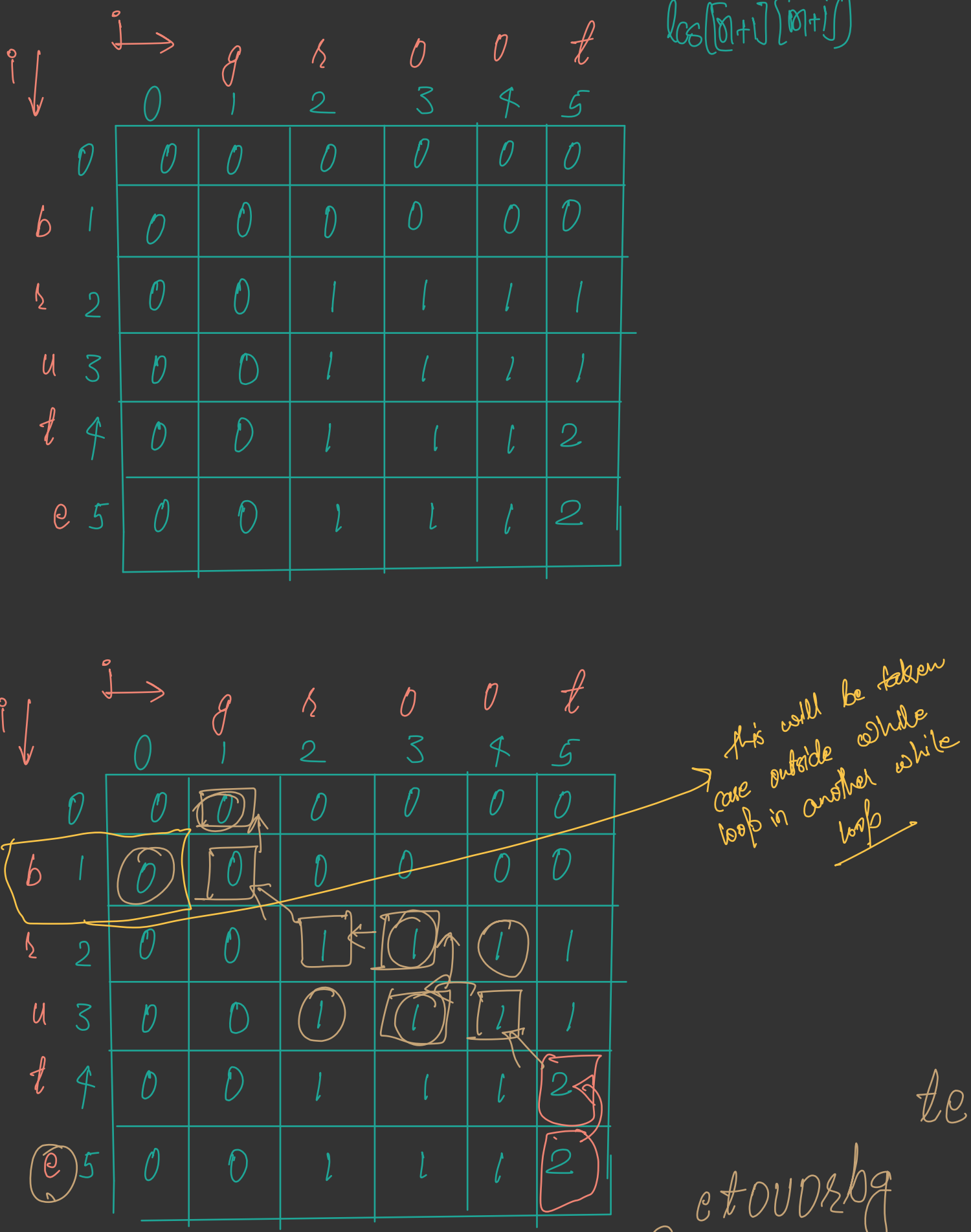
groot

bgruoote

traverse both string if char matches take it once, else take both

operations = n+m − len(lcs)

string = ?


i
j
g k o o t
0 1 2 3 4 5
0 0 0 0 0 0 0
b 1 0 0 0 0 0 0
k 2 0 0 1 1 1 1
u 3 0 0 1 1 1 1
t 4 0 0 1 1 1 2
e 5 0 0 1 1 1 2
dp[i+1][j+1]
this will be taken care outside while loop in another while loop
ctoudzbg
gbsovote
len -> 8
te

```javascript
function shortestSupersequence(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n + 1)
        .fill()
        .map(() => Array(m + 1).fill(0));

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = 1 + dp[i - 1][j - 1];
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    let i=n; j=m;
    let ans = '';

    while(i>0 && j>0) {
        if(s1[i-1] === s2[j-1]) {
            ans += s1[i-1];
            i--;
            j--;
        } else if(dp[i-1][j] > dp[i][j-1]) {
            ans += s1[i-1];
            i--;
        } else {
            ans += s2[j-1];
            j--;
        }
    }

    while(i>0) {
        ans += s1[i-1];
        i--;
    }

    while(j>0) {
        ans += s2[j-1];
        j--;
    }

    return ans.split('').reverse().join('');
}

const s1 = "brute";
const s2 = "groot";
shortestSupersequence(s1, s2); //? bgsoooote
```

only one will run

tabulation

Shortest Common Supersequence

TA Contributed by **TanishkTonk**
Last Updated: 23 Feb, 2023

Hard 0/120

Share 50 upvotes

Problem Statement Suggest Edit

Given two strings, 'A' and 'B'. Return the shortest supersequence string 'S', containing both 'A' and 'B' as its subsequences. If there are multiple answers, return any of them.
Note: A string 's' is a subsequence of string 't' if deleting some number of characters from 't' (possibly 0) results in the string 's'.
For Example:

```
Suppose 'A' = "brute", and 'B' = "groot"

The shortest supersequence will be "bgruoote". As shown
below, it contains both 'A' and 'B' as subsequences.

A   A A     A A
b g r u o o t e
  B B   B B B

It can be proved that the length of supersequence for this
input cannot be less than 8. So the output will be bgruoote.
```

---

lec 33 → distinct subsequences | 1D array optimisation technique.

concept → string matching

s1 = babgbag s2 = bag

ans = 5

babgbag

babgbag

babgbag

babgbag

babgbag

Try out all possible ways

Recursion

① express everything in (i, j)

② explore all possibilities on that index

③ return summation of all possibilities

s1 = babgbag (i, n, n-1)

s2 = bag (j, m, m-1)

start call

f(n-1, m-1) → number of distinct subsequences of s2[0....j] in s1[0.....i]

f(i,j) {
if(j<0) return 1;
if(i<0) return 0;

if(s1[i] == s2[j])
return f(i-1,j-1) + f(i-1,j)
else
return f(i-1,j)
}

TC → exponential
s1 → 2^n
s2 → 2^m
SC → O(N+M)

memoization
changing params → i j
dp[N][M]

TC → O(N×M)
SC → O(N×M) + O(N+M)
ASS
auxiliary stack space

base case
babgbag bag
-1 ↑ i
j ↑

if j == -1
↳ everything is matched
if all chars of s2 matched, return 1
if j>0 & i<0 → match not found
return 0

115. Distinct Subsequences

Hard 5.2K 199

Apple Google Amazon ...

Given two strings `s` and `t`, return *the number of distinct **subsequences** of `s` which equals `t`.*

The test cases are generated so that the answer fits on a 32-bit signed integer.

Example 1:

```
Input: s = "rabbbit", t = "rabbit"
Output: 3
Explanation:
As shown below, there are 3 ways you can generate "rabbit" from s.
rabbbit
rabbbit
rabbbit
```

```javascript
function f(i,j,s1,s2) {
    if(j<0) return 1;
    if(i<0) return 0;

    if(s1[i] === s2[j]) {
        return f(i-1, j-1, s1, s2) + f(i-1, j, s1, s2);
    } else {
        return f(i-1, j, s1, s2);
    }
}

var numDistinct = function (s1, s2) {
    let n = s1.length;
    let m = s2.length;
    return f(n-1,m-1,s1,s2)
};

const s1 = "babgbag";
const s2 = "bag";
numDistinct(s1, s2); //?5
```

Recursion

```javascript
function f(i, j, s1, s2, dp) {
    if (j < 0) return 1;
    if (i < 0) return 0;

    if (dp[i][j] != -1) return dp[i][j];

    if (s1[i] === s2[j]) {
        return (dp[i][j] = f(i - 1, j - 1, s1, s2, dp) + f(i - 1, j, s1, s2, dp));
    } else {
        return (dp[i][j] = f(i - 1, j, s1, s2, dp));
    }
}

var numDistinct = function (s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n)
        .fill()
        .map(() => Array(m).fill(-1));
    return f(n - 1, m - 1, s1, s2, dp);
};

const s1 = "babgbag";
const s2 = "bag";
numDistinct(s1, s2); //?5
```

Memoization

Tabulation — bottom-up

1. declare dp array
2. write base cases
3. write loops for changing params
4. copy the recurrence

dp[N][M]

shifting the index by 1 to right

1. dp[N+1][M+1]
2. for (i=0 to n) dp[i][0] = 1
   for (j=0 to m) dp[0][j] = 0
3. for (i=1 to n)
   for (j=1 to m)
4. if (s1[i-1] == s2[j-1])
   dp[i][j] = dp[i-1][j-1] + dp[i-1][j]
   else
   dp[i][j] = dp[i-1][j]

return dp[n][m]

```
var numDistinct = function (s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n+1)
        .fill()
        .map(() => Array(m+1).fill(0));

    for(let i=0; i<=n; i++) dp[i][0] = 1;
    // for(let j=1; j<=m; j++) dp[0][j] = 0;
    ↪ because dp is initialised with 0 already
    for(let i=1; i<=n; i++) {
        for(let j=1; j<=m; j++) {
            if(s1[i-1] === s2[j-1]) {
                dp[i][j] = dp[i-1][j-1] + dp[i-1][j]
            } else {
                dp[i][j] = dp[i-1][j];
            }
        }
    }

    return dp[n][m];
};

const s1 = "babgbag";
const s2 = "bag";
numDistinct(s1, s2); //? 5
```

Tabulation

```
var numDistinct = function (s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(0);
    let cur = Array(m+1).fill(0);

    prev[0] = cur[0] = 1;

    for(let i=1; i<=n; i++) {
        for(let j=1; j<=m; j++) {
            if(s1[i-1] === s2[j-1]) {
                cur[j] = prev[j-1] + prev[j]
            } else {
                cur[j] = prev[j];
            }
        }
        prev = [...cur]
    }

    return prev[m];
};

const s1 = "babgbag";
const s2 = "bag";
numDistinct(s1, s2); //? 5
```

space optimized

```
var numDistinct = function (s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(0);

    prev[0] =  1;

    for(let i=1; i<=n; i++) {
        for(let j=m; j>=1; j--) {
            if(s1[i-1] === s2[j-1]) {
                prev[j] = prev[j-1] + prev[j]
            }
        }
    }

    return prev[m];
};

const s1 = "babgbag";
const s2 = "bag";
numDistinct(s1, s2); //?5
```

1D - optimized

lec 34 → Edit Distance | 1D Array optimised

Problem Statement Suggest Edit

You are given two strings 'S' and 'T' of lengths 'N' and 'M' respectively. Find the "Edit Distance" between the strings.
Edit Distance of two strings is the minimum number of steps required to make one string equal to the other. In order to do so, you can perform the following three operations:

```
1. Delete a character
2. Replace a character with another one
3. Insert a character
```

Note:

```
Strings don't contain spaces in between.
```

Sample Input 1 :

```
abc
dc
```

Sample Output 1 :

```
2
```

Explanation For Sample Input 1 :

```
In 2 operations we can make the string T to look like string
S. First, insert the character 'a' to string T, which makes
it "adc".

And secondly, replace the character 'd' of the string T with
'b' from the string S. This would make string T to "abc"
which is also the string S. Hence, the minimum distance.
```

① insert
② remove
③ replace

M1 → delete all from s1 → n
insert all from s2 to s1 → m
total operations (n+m)

f(n-1, m-1) → find min operations to convert s1[0....i] to s2[0....j]

intuition

String matching

→ insert of the same character
→ delete & try finding somewhere else
→ replace & match

insertion

horse s — no match so insert s

sos

1 + f(i, j-1)

after insertion j matches with the inserted char, so j = j-1 & i stayed same

replace

horse sos

doesn't match, update e to s

i = i-1
j = j-1

1 + f(i-1, j-1)

remove/delete

horse sos

doesn't match, delete e,

i = i-1
j = j

1 + f(i-1, j)

matches.

$0 + f(i-1, j-1)$

Base case

i or j exhausted

s1 get exhausted

we need to insert remaining chars from s2

s2 get exhausted

-1 means match already found

s1 = "hos" s2 = ""

to convert s1 to s2

min & delete operations required

f(i,j){

if(i<0) return j+1;

if(j<0) return i+1;

if(s1[i]==s2[j]) return 0+f(i-1,j-1)

return Math.min(1+f(i,j-1)
1+f(i-1,j),
1+f(i-1,j-1))

}

TC → exponential

SC → O(N+M)

Memoization

changing params i j

N M

dp[N][M]

TC → O(N×M)

SC → O(N×M) + O(N+M)

ASS

```
function f(i, j, s1, s2) {
    if (i < 0) return j+1;
    if (j < 0) return i+1;

    if (s1[i] === s2[j]) {
        return 0 + f(i-1, j-1, s1, s2)
    } else {
        return Math.min((1+f(i, j-1, s1, s2)), (1+f(i-1, j, s1, s2)), (1+f(i-1, j-1, s1, s2)));
    }
}

function editDistance(s1, s2) {
    let n = s1.length;
    let m = s2.length;

    return f(n-1, m-1, s1, s2);
};

const s1 = "abc";
const s2 = "dc";
editDistance(s1, s2); //? 2
```

Recursion

```
function f(i, j, s1, s2, dp) {
    if (i < 0) return j + 1;
    if (j < 0) return i + 1;

    if (s1[i] === s2[j]) {
        return (dp[i][j] = 0 + f(i - 1, j - 1, s1, s2, dp));
    } else {
        return (dp[i][j] = Math.min(
            1 + f(i, j - 1, s1, s2, dp),
            1 + f(i - 1, j, s1, s2, dp),
            1 + f(i - 1, j - 1, s1, s2, dp)
        ));
    }
}

function editDistance(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n)
        .fill()
        .map(() => Array(m).fill(-1));

    return f(n - 1, m - 1, s1, s2, dp);
}

const s1 = "abc";
const s2 = "dc";
editDistance(s1, s2); //? 2
```

Memoization

Tabulation

```
function editDistance(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n+1)
        .fill()
        .map(() => Array(m+1).fill(0));

    for(let i=0; i<=n; i++) dp[i][0] = i;
    for(let j=0; j<=m; j++) dp[0][j] = j;

    for(let i=1; i<=n; i++) {
        for(let j=1; j<=m; j++) {
            if (s1[i-1] === s2[j-1]) {
                dp[i][j] = 0 + dp[i - 1][j-1];
            } else {
                dp[i][j] = Math.min(
                    1 + dp[i][j-1],
                    1 + dp[i - 1][j],
                    1 + dp[i - 1][j-1]
                );
            }
        }
    }

    return dp[n][m];
}

const s1 = "abc";
const s2 = "dc";
editDistance(s1, s2); //? 2
```

Tabulation

Shifting index to right by 1

```
function f(i, j, s1, s2, dp) {
    if (i <= 0) return j + 1;
    if (j <= 0) return i + 1;

    if (s1[i-1] === s2[j-1]) {
        return (dp[i][j] = 0 + f(i - 1, j - 1, s1, s2, dp));
    } else {
        return (dp[i][j] = Math.min(
            1 + f(i, j - 1, s1, s2, dp),
            1 + f(i - 1, j, s1, s2, dp),
            1 + f(i - 1, j - 1, s1, s2, dp)
        ));
    }
}

function editDistance(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n+1)
        .fill()
        .map(() => Array(m+1).fill(-1));

    return f(n - 1, m - 1, s1, s2, dp);
}
```

```
function editDistance(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(0);
    let cur = Array(m+1).fill(0);

    for(let j=0; j<=m; j++) prev[j] = j;

    for(let i=1; i<=n; i++) {
        cur[0] = i;
        for(let j=1; j<=m; j++) {
            if (s1[i-1] === s2[j-1]) {
                cur[j] = 0 + prev[j-1];
            } else {
                cur[j] = Math.min(
                    1 + cur[j-1],
                    1 + prev[j],
                    1 + prev[j-1]
                );
            }
        }
        prev = [...cur]
    }

    return prev[m];
}

const s1 = "abc";
const s2 = "dc";
editDistance(s1, s2); //? 2
```

space optimized

lec 35 -> wildcard matching

? -> matches with single character

* -> matches with sequence of length 0 or more

s1 = "?ay" true
s2 = "say"

s1 = ab*cd
s2 = abdefcd true

s1 = **abcd
s2 = abcd true

s1 = ab?d
s2 = abcc false

input
- s1 -> lowercase, ?, *
- s2 -> lowercase

question -> check whether s1 matches with s2 or not

TC -> exponential
SC -> O(N+M) ASS

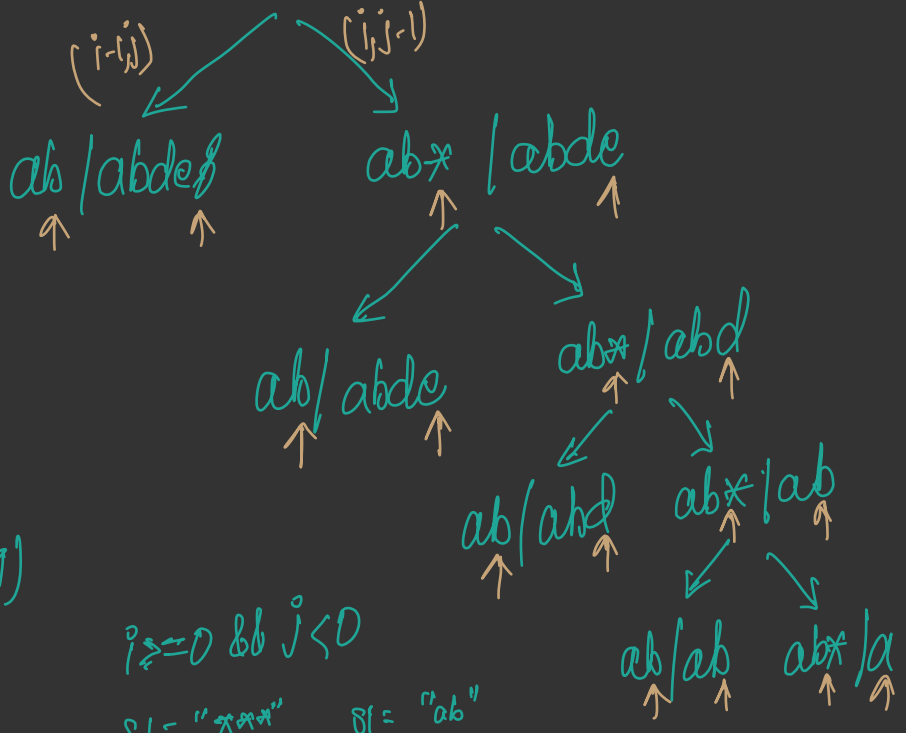
Recursion

① express in (i,j)
② explore all comparisons
③ out of all comparisons, if anyone can, return true

```
f(i,j)
{ if(i<0 && j<0) return true;
  if(i<0 && j>=0) return false;
if(i>=0 && j<0) ..... code .....
if(s1[i]==s2[j] || s1[i]=='?')
    return f(i-1,j-1)
if(s1[i]=='*')
    return f(i-1,j) or f(i,j-1)
return false;
}
```

for * case

ex ab* abdef



i>=0 && j<0

s1 = "***" s2 = "" true

s1 = "ab" s2 = "" false

Memoization

i j

n-1 m-1

dp[N][M]

TC → O(N×M)

SC → O(N×M) + O(N+M)

```
function f(i, j, s1, s2) {
    if (i < 0 && j<0) return true;
    if (i < 0 && j>=0) return false;
    if(i>=0 && j<0) {
        for(let x=i; x>=0; x--) {
            if(s1[x] != '*') return false;
        }
        return true;
    }

    if(s1[i] === s2[j] || s1[i] == '?') return f(i-1, j-1, s1, s2);
    if(s1[i] === '*') return f(i-1, j, s1, s2) || f(i, j-1, s1, s2);

    return false;
}

function wildcardMatching(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    return f(n-1, m-1, s1, s2);
}

const s1 = "*ay";
const s2 = "ray";
wildcardMatching(s1, s2); //? true
```

Recursion

```
function f(i, j, s1, s2, dp) {
    if (i < 0 && j<0) return true;
    if (i < 0 && j>=0) return false;
    if(i>=0 && j<0) {
        for(let x=i; x>=0; x--) {
            if(s1[x] != '*') return false;
        }
        return true;
    }

    if(dp[i][j] != -1) return dp[i][j];

    if(s1[i] === s2[j] || s1[i] == '?') return dp[i][j] = f(i-1, j-1, s1, s2, dp);
    if(s1[i] === '*') return dp[i][j] = f(i - 1, j, s1, s2, dp) || f(i, j - 1, s1, s2, dp);

    return dp[i][j] = false;
}

function wildcardMatching(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n).fill().map(() => Array(m).fill(-1));
    return f(n-1, m-1, s1, s2, dp);
}

const s1 = "*ay";
const s2 = "ray";
wildcardMatching(s1, s2); //? true
```

memoization

Tabulation

```
function wildcardMatching(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n + 1)
        .fill()
        .map(() => Array(m + 1).fill(false));

    dp[0][0] = true;
    for (let j = 1; j <= m; j++) dp[0][j] = false;
    for (let i = 1; i <= n; i++) {
        let flag = true;
        for (let x = i; x >= 1; x--) {
            if (s1[x - 1] != "*") flag = false;
        }
        dp[i][0] = flag;
    }

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] === s2[j - 1] || s1[i - 1] == "?")
                dp[i][j] = dp[i-1][j-1];
            else if (s1[i - 1] === "*")
                dp[i][j] =
                    dp[i-1][j] || dp[i][j-1];
            else {
                dp[i][j] = false;
            }
        }
    }

    return dp[n][m];
}

const s1 = "*ay";
const s2 = "ray";
wildcardMatching(s1, s2); //? true
```

Tabulation

```
function f(i, j, s1, s2, dp) {
    if (i == 0 && j == 0) return true;
    if (i == 0 && j > 0) return false;
    if (i > 0 && j == 0) {
        for (let x = i; x >= 1; x--) {
            if (s1[x - 1] != "*") return false;
        }
        return true;
    }

    if (dp[i][j] != -1) return dp[i][j];

    if (s1[i - 1] === s2[j - 1] || s1[i - 1] == "?")
        return (dp[i][j] = f(i - 1, j - 1, s1, s2, dp));
    if (s1[i - 1] === "*")
        return (dp[i][j] = f(i - 1, j, s1, s2, dp) || f(i, j - 1, s1, s2, dp));

    return (dp[i][j] = false);
}

function wildcardMatching(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n + 1)
        .fill()
        .map(() => Array(m + 1).fill(-1));
    return f(n, m, s1, s2, dp);
}

const s1 = "*ay";
const s2 = "ray";
wildcardMatching(s1, s2); //? true
```

1 based indexing

```javascript
function wildcardMatching(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(false);
    let cur = Array(m+1).fill(false);

    prev[0] = true;
    for (let j = 1; j <= m; j++) prev[j] = false;

    for (let i = 1; i <= n; i++) {
        // cur is the current's row's column
        // and that cur's 0th row has to be assigned everytime
        let flag = true;
        for (let x = i; x >= 1; x--) {
            if (s1[x - 1] != "*") flag = false;
        }
        cur[0] = flag;

        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] === s2[j - 1] || s1[i - 1] == "?")
                cur[j] = prev[j-1];
            else if (s1[i - 1] === "*")
                cur[j] =
                    prev[j] || cur[j-1];
            else {
                cur[j] = false;
            }
        }

        prev = [...cur];
    }

    return prev[m];
}

const s1 = "*ay";
const s2 = "ray";
wildcardMatching(s1, s2); //? true
```

space optimized

---

lec 86 best time to buy and sell stocks | DP on stocks

arr[] → [7, 1, 5, 3, 6, 4] n = 6

buy → 1
sell → 6
profit ⇒ 5

intuition

if you are selling on ith day, you buy on the min price from 1st → (i-1)

code

```
min = a[0], profit = 0
for(i=1; i<n; i++)
  { cost = a[i] - min;
    profit = max(profit, cost);
    min = min(min, a[i])
  }
return profit;
```

T(n) → O(N)
S(n) → O(1)

```javascript
function maxProfit(arr) {
    let n = arr.length;
    let min = arr[0];
    let profit = 0;

    for(let i=1; i<n; i++){
        let cost = arr[i] - min;
        profit = Math.max(profit, cost);
        min = Math.min(min, arr[i]);
    }

    return profit;
}

const arr = [7,1,5,3,6,4]
maxProfit(arr); //? 5
```