Analysis of Algorithms

V. Adamchik

CSCI 570

Lecture 6

University of Southern California

Spring 2023

Dynamic Programming

Reading: chapter 6

Review

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved by the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

$T(n) = \Theta(n^3)$

1. $T(n) = 16\, T(n/4) + 5\, n^3 + \log n$
2. $T(n) = 4\, T(n/2) + n^2 \log n$ $T(n) = \Theta(n^2 \log^2 n)$
3. $T(n) = 4\, T(n/8) - n^2$ NA
4. $T(n) = 2^n\, T(n/2) + n$ NA
5. $T(n) = 0.2\, T(n/2) + n \log n$ NA

≥ 1

Review

Design a new Mergesort algorithm in which instead of splitting the input array in half we split it in the ratio 1:3.

Write down the recurrence relation for the number of comparisons.

What is the runtime complexity of this algorithm?

$n/4$	$3n/4$

① $T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + O(n)$

② $T(n) = \Theta(\cancel{?})$

$n \log n$

can you use the master theorem?

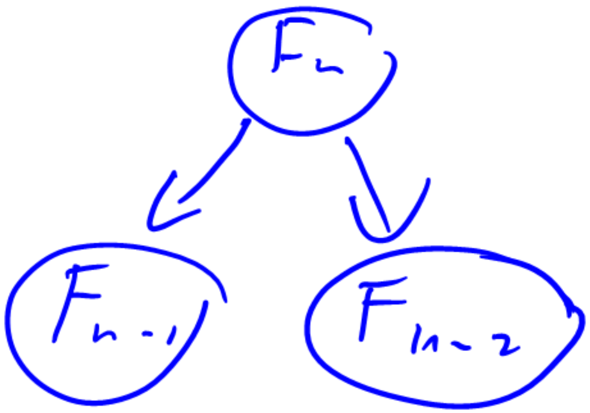
NO

REVIEW QUESTIONS

1. (T/F) For a divide-and-conquer algorithm, it is possible that the dividing step takes asymptotically longer time than the combining step. LL
2. (T/F) A divide-and-conquer algorithm acting on an input size of n can have a lower bound less than $\Theta(n \log n)$. binary search
3. (T/F) There exist some problems that can be efficiently solved by a divide-and-conquer algorithm but cannot be solved by a greedy algorithm. sort
4. (T/F) It is possible for a divide-and-conquer algorithm to have an exponential runtime.
5. (T/F) A divide-and-conquer algorithm is always recursive.
6. (T/F) The master theorem can be applied to the following recurrence: $T(n) = 1.2\,T(n/2) + n$.
7. (T/F) The master theorem can be applied to the following recurrence: $T(n) = 9\,T(n/3) - n^2 \log n + n$.
8. (T/F) Karatsuba's algorithm reduces the number of multiplications from four to three.
9. (T/F) The runtime complexity of mergesort can be asymptotically improved by recursively splitting an array into three parts (rather than into two parts).

Fibonacci Numbers

Fibonacci number F_n is defined as the sum of two previous Fibonacci numbers:



$$F_n = F_{n-1} + F_{n-2}$$

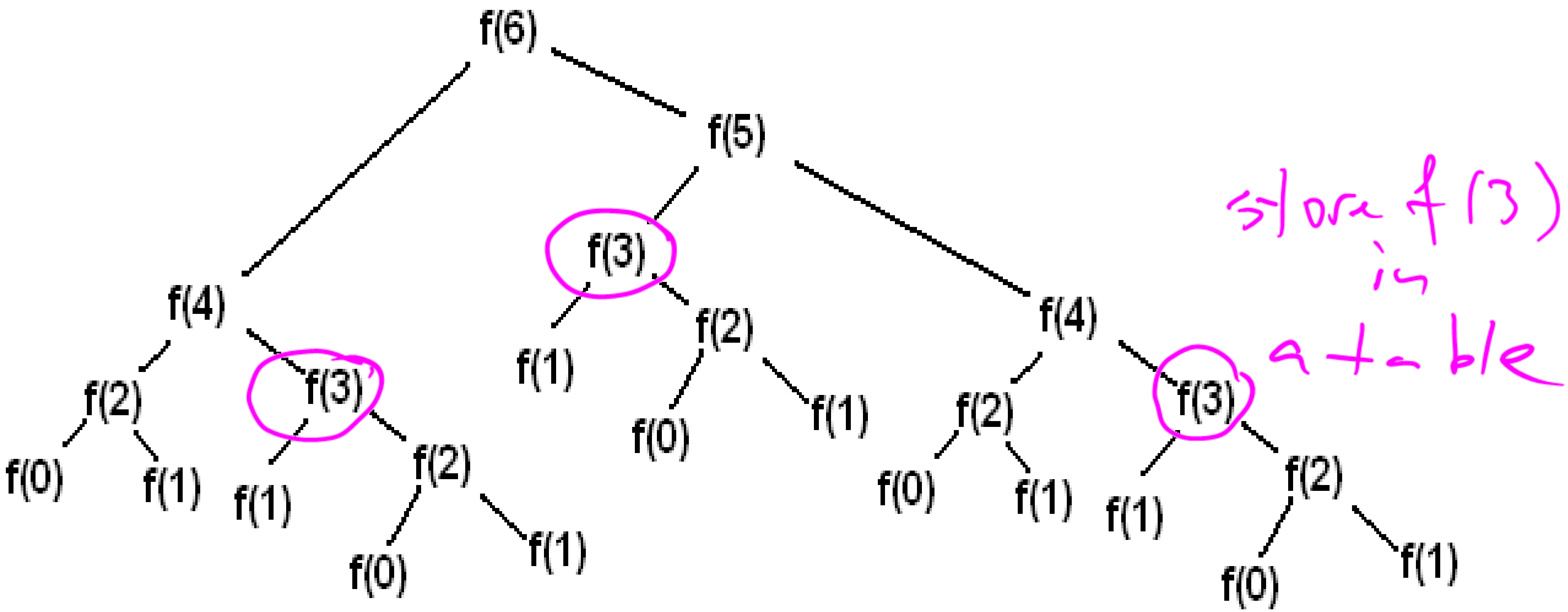
$$F_0 = F_1 = 1$$

Design a divide & conquer algorithm to compute Fibonacci numbers. What is its runtime complexity?

$$T(n) = T(n-1) + T(n-2) + O(n)$$

exponential

Overlapping Subproblems



Fibonacci Numbers: $F_n = F_{n-1} + F_{n-2}$

Memoization

```
int  table [50];    //initialize to zero
table[0] = table[1] = 1;

int fib(int  n)
{
        if  (table[n] != 0) return  table[n];
        else
        table[n] = fib(n-1) + fib(n-2);

        return  table[n];
}
```

$T(n) = (n-1) + (n-2) + (n-3) + \ldots = O(n^2)$

$\Theta(1)$

Runtime complexity?

$T(n) = T(n-1) + T(n-2) + O(n)$

$T(n) = \Theta(n^2)$

Tabulation → 570

```
int  table [n];

void fib(int n)
{
        table[0] = table[1] = 1;
        for(int k = 2; k < n; k++)
            table[k] = table[k-1] + table[k-2];

        return;
}
```

Two Approaches

```
int  table [n];
table[0] = table[1] = 1;
int  fib(int  n)
{
   if  (table[n] != 0)
      return  table[n];
   else
      table[n] = fib(n-1) + fib(n-2);
   return  table[n];
}
```

n
n-1
⋮
2
1

n
⋮
2
1

```
int  table [n];

int[] fib(int n)
{
   table[0] = table[1] = 1;
   for(int k = 2; k < n; k++)
      table[k]=table[k-1]+table[k-2];

   return  table;
}
```

n
n-1
⋮
1

Memoization:
a top-down approach.

Tabulation:
a bottom-up approach.

cs 570

Dynamic Programming

General approach: in order to solve a larger problem, we solve smaller subproblems and store their values in a table.

DP is applicable when the subproblems are greatly overlap. Compare with Mergesort.

DP is not greedy either. DP tries <u>every</u> choice before solving the problem. It is much more expensive than greedy.

DP can be implemented by means of memoization or tabulation.

Dynamic Programming

Optimal substructure means that the solution can be obtained by the combination of optimal solutions to its subproblems. Such optimal substructures are usually described recursively.

must

Overlapping subproblems means that the space of subproblems must be small, so an algorithm solving the problem should solve the same subproblems over and over again.

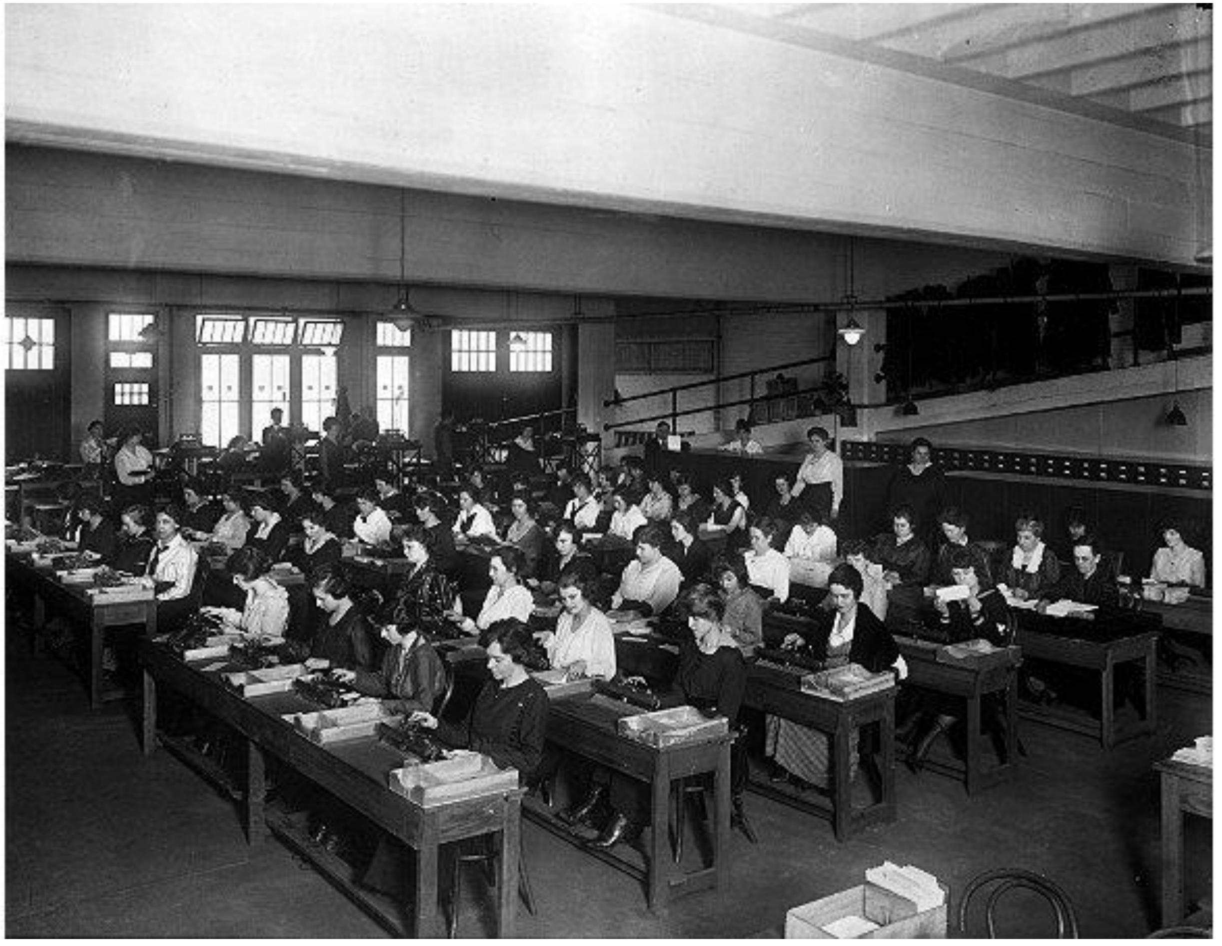
Dynamic Programming

The term dynamic programming was originally used in the 1950s by Richard Bellman.

The term computer (dated from 1613) meant a person performing mathematical calculations.

In the 30-50s those early computers were mostly *women* who used painstaking calculations on paper and later punch cards.

The earliest human computers



Who put a man to the moon?

0-1 Knapsack Problem

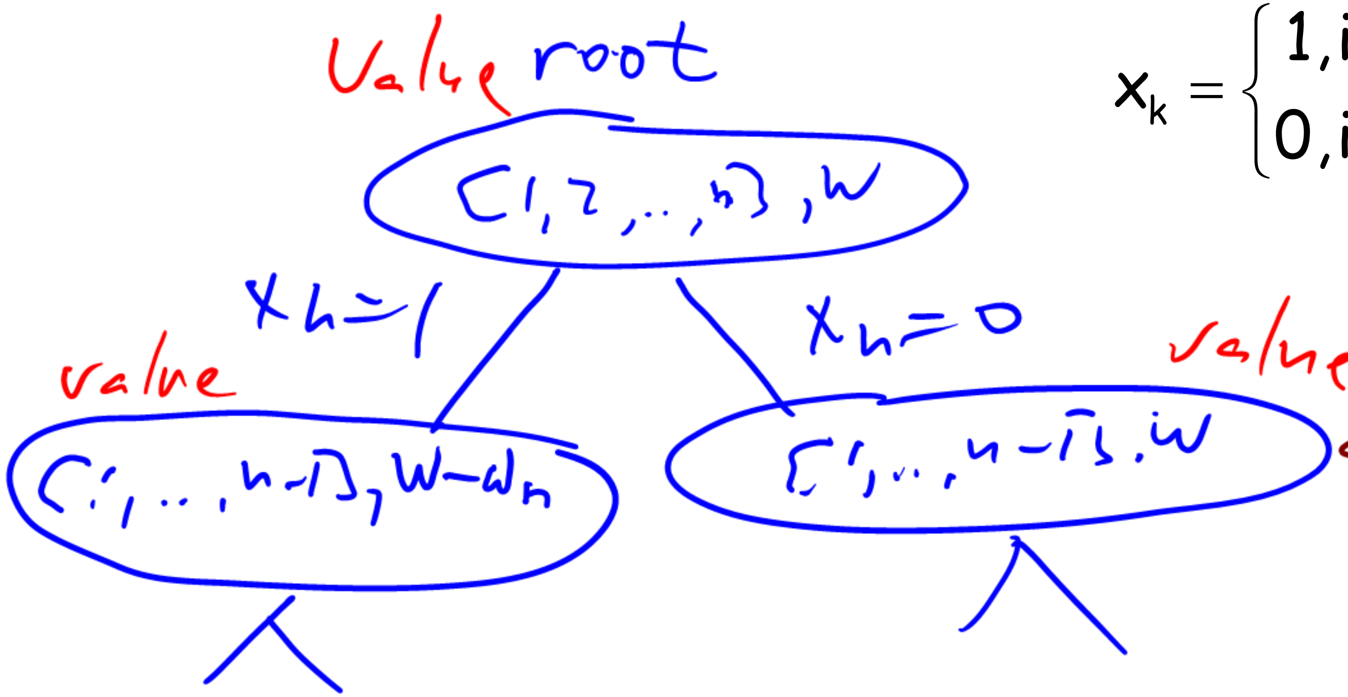
Given a set of n unbreakable *unique* items, each with a weight w_k and a value v_k, determine the subset of items such that the total weight is less or equal to a given knapsack capacity W and the total value is as large as possible.

Fractional knapsack - by greedy algo.

Brute-force, runtime $O(2^n)$

Decision Tree

$$x_k = \begin{cases} 1, \text{item k selected} \\ 0, \text{item k not selected} \end{cases}$$



subproblem

OPT[k, x]

k is # of items, $0 \le k \le n$

x is a capacity of, $0 \le x \le W$

Subproblems

Let OPT[K, X] be the max value achievable using a knapsack of capacity X and K items.

Our choices:

① $x_K = 0$, $OPT[K, X] = OPT[K-1, X]$

② $x_K = 1$, $OPT[K, X] = v_K + OPT[K-1, X-w_K]$

Recurrence Formula

$$\overbrace{OPT[k,x]}^{O(1)} = \overbrace{MAX}^{O(1)}\left(\overbrace{OPT[k-1, x]}^{O(1)},\ \underbrace{V_k + OPT[k-1, x-w_k]}_{O(1)} \right)$$

entry in a table

Base cases:

$OPT[k,x] = 0$, if $k=0$ or $x=0$

$OPT[k,x] = OPT[k-1,x]$, if $w_k > x$

Tracing the Algorithm

$n = 4$, $W = 5$

(weight, value) = (2,3), (3,4), (4,5), (5,6)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	? 0	? 3	3	3	3
1,2	0	0	? 3	? 4	4	? 7
1,2,3	0	0	3	4	? 5	7
1,2,3,4	0	0	3	4	5	7

knapsack capacity

items

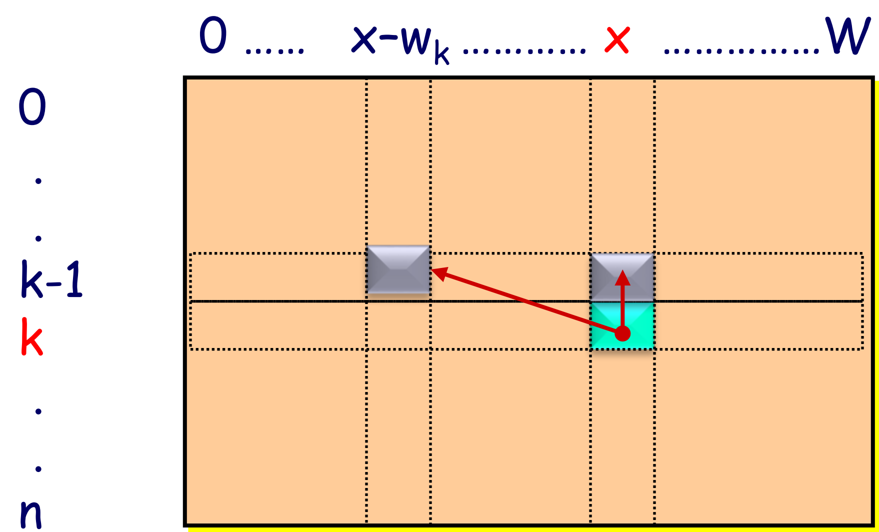
OPT[n, W]

table of values

Pseudo-code

```
int knapsack(int W, int w[], int v[], int n) {   <-- input
   int Opt[n+1][W+1];                                 ?
   for (k = 0; k <= n; k++) {
      for (x = 0; x <= W; x++) {
         if (k==0 || x==0) Opt[k][x] = 0;
         if (w[x] > x) Opt[k][x] = Opt[k-1][x];
         else
           Opt[k][x] = max( v[k] + Opt[k-1][x - w[k-1]],
                                        Opt[k-1][x] );
      }
   }
   return Opt[n][W];   solution
}
```

Complexity



$O(n \cdot W)$

$O(n \cdot W) \cdot O(1)$

Runtime Complexity? tablesize x the work per cell

Space Complexity? table size

$O(n \cdot W)$

Pseudo-Polynomial Runtime

<u>Definition</u>. A numeric algorithm runs in pseudo-polynomial time if its running time is polynomial in the numeric value of the input but is exponential in the length of the input.

$W = \max(w_1, .., w_n)$, $V = \max(v_1, .., v_n)$

0-1 Knapsack is pseudo-polynomial algorithm, T(n) = Θ(n·W)

Input size: $O(\log W + n \cdot \log W + \cancel{n \cdot (\log V + \log n})$)

Runtime: $O(n \cdot W)$

Input size: $O(n \cdot \log W)$

Actual Runtime: $O(n \cdot 2^{\text{input size of } w})$ in the number of bits

exponential

How would you find the actual items?

The table built in the algorithm does not show the optimal items, but only the maximum value. How do we find which items give us that optimal value?

n = 4, W = 5

(weight, value) = (2, 3), (3, 4), ~~(4, 5)~~, ~~(5, 6)~~

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

DP Approach

solve using the following four steps:

1. Define (in plain English) subproblems to be solved.
2. Write the recurrence relation for subproblems.
3. Write pseudo-code to compute the optimal value.
4. Compute the runtime of the above DP algorithm in terms of the input size.

Discussion Problem 1

no repetitions!

You are to compute the minimum number of coins needed to make change for a given amount m. Assume that we have an unlimited supply of coins. All denominations d_k are sorted in ascending order:

$$1 = d_1 < d_2 < \dots < d_n$$!!

step 1.

Let $OPT[k, x]$ be the min number of coins to represent x $(0 \le x \le m)$ using first k $(1 \le k \le n)$ denominations.

step 2. Recurrence for $OPT[k, x]$

$$OPT[k,x] = \min\left(OPT[k-1, x],\ 1 + OPT[\underline{k-1}, x - d_k] \right)$$

no repetitions

Base cases:

$OPT[k, 0] = 0$

$OPT[1, x] = x$

step 3. Pseudo-code: DIY

step 4. Runtime?

is it polynomial?

$O(n \cdot m)$ not input size

NO

Longest Common Subsequence

DNA

We are given string S_1 of length n, and string S_2 of length m.

Our goal is to produce their longest common subsequence. (LCS)

A *subsequence* is a subset of elements in the sequence taken in order (with strictly increasing indexes.) Or you may think as removing some characters from one string to get another.

Note, a subsequence is not a substring.

diff in Unix

Intuition

A B A (Z D) C $\leftarrow s_1(i_1, \ldots, j_1)$

B A (C B) A D $\leftarrow s_2(i_2, \ldots, j_n)$

input 0 1 2 .. n

① $s_1(0 \ldots i)$ prefix

② $s_1(i \ldots n)$ suffix

③ $s_1(i \ldots j)$

A, AB, ABA, ..

C, DC, ZDC, ..

Subproblems

Let $OPT[i,j]$ be the max length of the LCS of $S_1[0..i]$ and $S_2[0..j]$

$0 \le i \le n$ $\qquad$ $0 \le j \le m$

Choices:

① $S_1[i] = S_2[j]$ (last characters)

$$OPT[i,j] = 1 + OPT[i-1, j-1]$$

② $S_1[i] \neq S_2[j]$

$$OPT[i,j] = \max(OPT[i-1,j], OPT[i,j-1])$$

Recurrence

Combine two cases:

$$OPT[i,j] = \max\Big(1 + OPT[i-1, j-1],$$
$$\max\big(OPT[i-1, j],\ OPT[i, j-1]\big)\Big)$$

Base cases:

$OPT[0,j] = 0$, $OPT[i,0] = 0$

Runtime: $O(n \cdot m)$ is it polynomial? YES

Example

S = ABAZDC
T = BACBAD

		B	A	C	B	A	D
	0	0	0	0	0	0	0
A	0	0	? 1	1	1	1	1
B	0	? 1	1	1	2	2	2
A	0						
Z	0						
D	0						
C	0						

BACB
AB

← answer

Pseudo-code

```
int LCS(char[] S1, int n, char[] S2, int m)
{
int table[n+1, m+1];
table[0…n, 0] = table[0, 0…m] = 0; //init
for(i = 1; i <= n; i++)
  for(j = 1; j <= m; j++)
      if (S1[i] == S2[j])  table[i, j] = 1 + table[i-1, j-1]        diagonal
      else
      table[i, j] = max(table[i, j-1], table[i-1, j]);

return table[n, m];  solution
}
```

How much space do we need?

		B	A	C	B	A	D
	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1
B	0	1	1	1	2	2	2
A	0	1	2	2	2	3	3
Z	0	1	2	2	2	3	3
D	0	1	2	2	2	3	4
C	0	1	2	3	3	3	4

2 rows
linear space!

How do we find the common sequence?

	0	0	0	0	0	0	0
	0	0	1	1	1	1	1
	0	1	1	1	2	2	2
	0	1	2	2	2	3	3
	0	1	2	2	2	3	3
	0	1	2	2	2	3	4
	0	1	2	3	3	3	4