4. Dynamic Programming (DP)

"programming" refers to "tabular" method, not to writing program code.

DP is a general, powerful alg. design technique for solving optimization problems

DP ≙ type of very smart exhaustive search that can be applied if the problem can be "subdivided" into overlapping subproblems.

Exmpl. Fibonacci numbers: $f(1) = f(2) = 1$

$f(n) = f(n-1) + f(n-2)$

⇒ 1, 1, 2, 3, 5, 8, 13, ...

Naive recursive way:

```
F(n)
  IF(n ≤ 2): f = 1
  ELSE  f = F(n-1) + F(n-2)
  return f
```

Exponential time!

Why: Recurrence Nr:

$T(n)$ = time to compute n-th Fibnr.

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

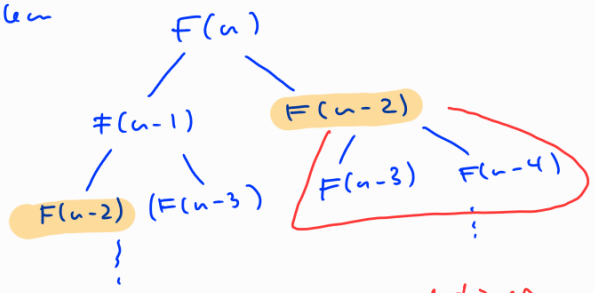
$\geq 2T(n-2)$

$\geq 2 \cdot 2\, T(n-4)$ ($T(n-1) \geq T(n-2)$)

$\vdots$

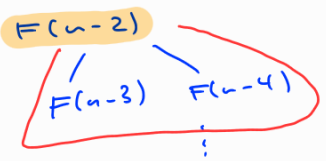
$\geq 2^{\frac{n}{2}} = O(2^n)$

"problem"



we compute F(k) several times, although it is enough to compute it ones

we can prune the entire search tree:



⇒ we must "memorize" the intermediate results.

```
memo = ∅        // dictionary

mF(n)
    IF (n ∈ memo) return memo[n]      // check if F[n] already computed.
    IF (n ≤ 2)   f = 1                } still the same lines.
    ELSE  f = mF(n-1) + mF(n-2)       }
    memo[n] = f
    return f
```

Runtime sketch: ⇒ mF(k) only recurses the first time it's called ∀ k
& memoized calls cost O(1)

the nr of non-memoized calls is n:
mF(1), mF(2), ... mF(k) ... mF(n)

⇒ runtime O(n) ! (goes even faster)

In general, the design of DP-Alg consists of the following steps:

Step 1: characterize structure of optimal solution
Step 2: Recursively define value of opt. solution
Step 3: Compute value of opt solution (typically bottom up)
Step 4: Construct opt solution from computed information

IF we only want to know the values of opt. sol. then 1-3 are sufficient.

4.1. The Floyd-Warshall Algorithm

Aim: Find shortest u-v-path $\forall\, u, v \in V(G)$.

Def: path $\langle v_0, \dots, v_k \rangle$ the vertices $v_1, \dots, v_{k-1}$ are inner vertices of path.

In what follows $V = \{1, 2, \dots, n\}$
$V_k = \{1, 2, \dots k\}$, $k \leq n$

Step 1: Characterize structure of shortest path in G with conservative weights

IDEA: let $i, j \in V$ & consider all i-j-path s.t all inner vertices are from V_k

Let p be one of such paths with minimum weight. = shortest i-j-path with all inner vertices in V_k (not necessarily shortest i-j-path in G)

Q: What is the relationship between p & shortest i-j-path with all inner vertices in V_{k-1}?

=> 2 cases:

- k is not inner vertex of p
  - => all inner vertices of p are in V_{k-1}
  - => shortest i-j path with all inner vertices in V_{k-1} is also a shortest i-j-path with all inner vertices in V_k

- k is inner vertex of p

=>

By lemma 3.1 "subpath of shortest paths are shortest paths"

P_1 is shortest i-k-path with inner vert. in V_{k-1}

P_2 is shortest k-j-path with inner vert. in V_{k-1}

=> this generalizes to: for shortest i-j-path P in G: all inner vertices are contained in $V_n = V$

IF u not part of P => P is shortest i-j path in G with inner vertices in V_{n-1}

u is part of P => P composed of shortest i-u path & u-j path with inner vertices in V_{n-1}.

Step 2: Recursively def values of shortest paths

- $d_{ij}^{(k)}$ = weight of shortest i-j-path with inner vertices from V_k

IF $\exists$ i-j-path & $k=0$ $\Rightarrow$ $(i,j) \in E$ & $d_{ij}^{(0)} = w(i,j)$

- $W_{ij} = \begin{cases} 0 & , i=j \\ w(i,j) & , i \neq j \wedge (i,j) \in E \\ \infty & , i \neq j \wedge (i,j) \notin E \end{cases}$ (gives matrix W)

Due to the discussion in (1):

$$d_{ij}^{(k)} = \begin{cases} W_{ij} & , k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right) & , k \geq 1 \end{cases}$$

$\Rightarrow$ Since for any path it's inner vertices are from V_n we obtain matrix $D^{(n)} = (d_{ij}^{(n)})$ with $d_{ij}^{(n)} = \delta(i,j)$ $\forall i,j \in V$

Step 3: Computing values of shortest paths

(Bottom up) (matrix W as above)

```
FLOYD-WARSHALL(W, n = |V|)
    D^0 = W
    FOR (k = 1 ... n) DO
        D^(k) = (d_ij^(k)) is new n x n matrix
        FOR (i = 1 ... n) DO
            FOR (j = 1 ... n) DO
                d_ij^(k) = min{ d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1) }
    Return D^(n)
```

Theorem 4.1 Floyd-Warshall algorithm correctly computes $\delta(i,j)$ $\forall i,j \in V$ for $G=(V,E)$ with conserv. weights in $\Theta(|V|^3)$ time

proof: Runtime: clear $\Theta(|V|^3)$
Correctness: discussion above ✓ □

(4) construct shortest i-j-paths $\forall i,j \in V$

// $\pi_{ij}^{(k)}$ = predecessor of j on shortest path from i to j with all inner vertices in V_k

Step 4: similar to $d_{ij}^{(k)}$ def:

$$\pi_{ij}^{(0)} = \begin{cases} NIL, & i = j \text{ or } w_{ij} = \infty \\ i, & i \neq j \text{ \& } w_{ij} < \infty \end{cases}$$

$k \geq 1$: Consider: $i \rightsquigarrow k \rightsquigarrow j$ (k is either part of this shortest i-j-path with inner vertices in V_k or not.

IF $k \neq j$ $\Rightarrow$ predecessor of j we choose is the same predessor of j we choose on shortest k-j-path with vertices in V_{k-1}

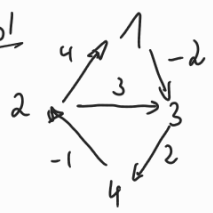
$k = j$ or k not part of this path. $\Rightarrow$ predecessor of j we choose the same predecessor of j we choose on shortest i-j path with all inner vertices in V_{k-1}

$$\Rightarrow \pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

[Exercise: understand + incorporate in pseudocode].

```
FLOYD-WARSHALL(W, n = |V|)
    D^0 = W
    FOR (k = 1 ... n) DO
        D^(k) = (d_ij^(k)) is new n x n matrix
        FOR (i = 1 ... n) DO
            FOR (j = 1 ... n) DO
                d_ij^(k) = min{ d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1) }
    Return D^(n)
```

Exmpl



$D^{(0)}$

	1	2	3	4
1	0	∞	-2	∞
2	4	0	3	∞
3	∞	∞	0	2
4	∞	-1	∞	0

D^{1}

	1	2	3	4
1	0	∞	-2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	∞	-1	∞	0

← shortest path cont. inner vertices from $V_1 = \{1\}$

(1) the first interesting case is $k=1, i=2, j=3$

∃ shorter path along 1

$\Rightarrow$ update $d_{23}^{(1)} = d_{21}^{(0)} + d_{13}^{(0)}$

$= 4 - 2$

$= 2$

(2) next interesting case $k=2, i=4, j=1$

so-far: $d_{41}^{(1)} = \infty$

→ but exist path from 4 to 1 with inner vertices from $V_2 = \{1,2\}$

$\Rightarrow d_{41}^{(2)} = d_{42}^{(1)} + d_{42}^{(1)}$

$= -1 + 4 = 3$

D^{2}

	1	2	3	4
1	0	∞	-2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	3	-1	1	0

next int. case: $k=2, i=4, j=3$

$d_{43}^{(2)} = d_{42}^{(1)} + d_{23}^{(1)} = -1 + 2 = 1$

D^{3}

	1	2	3	4
1	0	∞	-2	0
2	4	0	2	4
3	∞	∞	0	2
4	3	-1	1	0

next int case: $k=3, i=1, j=4$

$d_{14}^{(2)} = \infty$

$d_{14}^{(3)} = d_{13}^{(2)} + d_{34}^{(2)}$

$= -2 + 2 = 0$

next int case $k=3, i=2, j=4$

$d_{24}^{(2)} = \infty$

$d_{24}^{(3)} = d_{23}^{(2)} + d_{34}^{(2)}$

$= 2 + 2 = 4$

D^{4} = $D^{(n)}$

	1	2	3	4
1	0	-1	-2	0
2	4	0	2	4
3	5	-1	0	2
4	3	-1	1	0

next interesting case $k=4, i=1, j=2$

$d_{12}^{(3)} = \infty$

$d_{12}^{(4)} = d_{14}^{(3)} + d_{42}^{(3)}$

$= 0 - 1 = -1$

next case: $k=4, i=3, j=1$

$d_{31}^{(3)} = \infty$

$d_{31}^{(4)} = d_{34}^{(3)} + d_{41}^{(3)}$

$= 2 + 3 = 5$

$k=4, i=3, j=2$

$d_{32}^{(3)} = \infty$

$d_{32}^{(4)} = d_{34}^{(3)} + d_{42}^{(3)}$

$2 + -1 = 1$

4.2 The Longest Common Subsequence (LCS) problem

This is a classical problem in Bioinformatics, where one wants to understand how "close" DNA/protein sequences are.

= simply a string some alphabet.

There are several ways to address this problem. Here we consider one of them: LCS.

Def: Let $X = x_1 \cdots x_m$ & $Z = z_1 \cdots z_k$ be two strings (= sequences)

Z is a subsequence of X if

$\exists$ indices of X $i_1 \cdots i_k$ s.t. $i_1 < i_2 < \cdots < i_k$

& $z_j = x_{i_j}$

Exmpl: $X = A\,B\,C\,B\,D\,A\,B$, $Z = B\,C\,D\,B$

$\Rightarrow$ Z can be obtained from X by removing elements from X. & keep order of remaining elements.

IF Z is subsequence of X & Y $\Rightarrow$ Z is common subsequ. of X & Y.

Exmpl: $X = A\,B\,C\,B\,D\,A\,B$

$Y = B\,D\,C\,A\,B\,A$

$Z = BCA$, $Z' = BCBA$, $Z'' = BDAB$ are common subsequ. of X & Y.

Aim: Find longest common subsequ. (LCS) of X & Y.

Brute FORCE is not a good idea, Since $X = x_1 \cdots x_m$ has 2^m subsequences!

For $X = x_1 \cdots x_m$, define the i-th prefix X_i of X as $X_i = x_1 \cdots x_i$, $1 \le i \le m$ & put $X_0 = \varepsilon$ "empty sequ."

Step 1: Characterization of LCS.

Theorem 4.2 [Opt. substructure of LCS]

Let $X = x_1 \dots x_m$, $Y = y_1 \dots y_n$ be two sequences.
& $Z = z_1 \dots z_k$ be some LCS of X & Y.

(1) $x_m = y_n \Rightarrow z_k = x_m = y_n$ & z_{k-1} is LCS of x_{m-1} & y_{n-1}

(2) $x_m \neq y_n$ & $z_k \neq x_m \Rightarrow Z$ is LCS of x_{m-1} & Y

(3) $x_m \neq y_n$ & $z_k \neq y_n \Rightarrow Z$ is LCS of X & y_{n-1}.

Illustration: (1)

$X = A\ A\ B\ A\ C\ D$
$Y = A\ B\ B\ C\ D$
LCS $Z = A\ B\ C\ D$
(Z_3 = A B)

$\Rightarrow z_3$ is LCS of $X_5 = x_{m-1}$, $y_4 = y_{n-1}$, $m = 6, n = 5$

$\Rightarrow$ can reduce problem to find LCS of x_{m-1} & y_{n-1}

(2)

$X = A\ A\ B\ A\ C\ A$
$Y = A\ B\ B\ C\ D$
LCS $Z = A\ B\ C$

! this is independent of letter y_n
that is both cases may occur:
$z_k = y_n$ or $z_k \neq y_n$

But since z_k never "matches" x_n,
we can reduce the problem to find
LCS of x_{m-1} & Y

(3) analog to (2).

proof: [Thm 4.2]

(1) $x_m = y_n \Rightarrow z_k = x_n$ [otherwise if $z_k \neq x_n$ we can append x_n to $\underbrace{z_1 \dots z_k}_{\text{already CS}}$ & get larger subsequence $\lightning$ since z is LCS.]

$\Rightarrow x_m = x_n = z_k$

Now prefix z_{k-1} is a common subseqn. of x_{m-1} & y_{n-1}

to show: z_{k-1} is $\underline{\underline{L}}$CS of x_{m-1} & y_{n-1}

But this is clear, since if there is a longer one,

say W then we could append $z_k + W$ to obtain

a seqn. which is longer than z $\lightning$ z is LCS

(2) Clearly if $z_k \neq x_m \Rightarrow z$ is common subseq. of x_{m-1} & y.

to show z is $\underline{\underline{L}}$CS of x_{m-1} & y.

Again, if there is a longer common subseqn. W of x_{m-1} & y

then W is also a common subseq. of x & y,

but longer than z $\lightning$ z is LCS

(3) Analog to (2)

□

Step 2: recursive solution of LCS

By Thm 4.2: $x_m = y_n \Rightarrow$ Find LCS of X_{m-1} & Y_{n-1} & append $x_m = y_n$ to this LCS.

$x_m \neq y_n \Rightarrow$ Find LCS of x_{m-1} & y or x & y_{n-1} whichever of these is longer is an LCS of x & y

Let c_{ij} = length (# of letters) of LCS of prefixes X_i & Y_j

NOTE $x_0 = \varepsilon$, $y_0 = \varepsilon \Rightarrow c_{ij} = 0$, if $i = 0$ or $j = 0$

Recursive Formula:

$$c_{ij} = \begin{cases} 0 & \text{, if } i=0 \text{ or } j=0 \\ c_{i-1,j-1} + 1 & \text{, } i,j>0 \text{ \& } x_i = y_j \\ \max\{c_{i,j-1}, c_{i-1,j}\} & \text{, } i,j>0 \text{ \& } x_i \neq y_j \end{cases}$$

Step 3: Computing length of LCS.

Based on recursive formula for $c[i,j]$
we get for free an exponential-time recursive alg.
But DP can be used to get it in polynomial time.

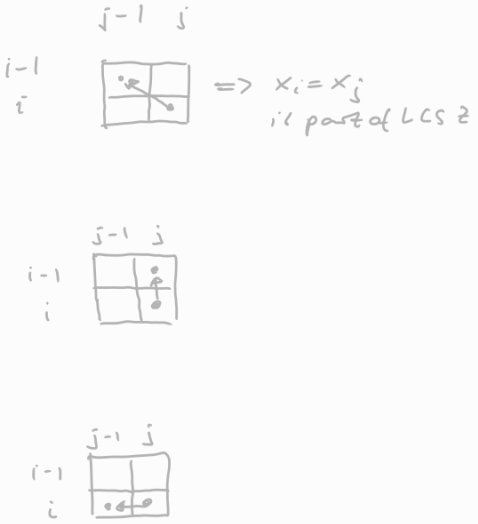
Let $X = x_1 \cdots x_m$ & $Y = y_1 \cdots y_n$

```
LCS(X,Y)
    Let b[1...m, 1...n] and c[0...m, 0...n] be new arrays   // b used to construct LCS via backtracking
                                                            // c stores length.
    FOR (i = 0...m) DO c[i,0] = 0
    FOR (j = 0...n) DO c[0,j] = 0

    FOR (i = 1...m) DO
        FOR (j = 1...n) DO
            IF (x_i = y_j)
                c[i,j] = c[i-1,j-1] + 1
                b[i,j] = "↖"
            ELSE IF (c[i-1,j] ≥ c[i,j-1])
                c[i,j] = c[i-1,j]
                b[i,j] = "↑"
            ELSE
                c[i,j] = c[i,j-1]
                b[i,j] = "←"

    RETURN c & b.
```

// c:



Step 4: construct LCS:

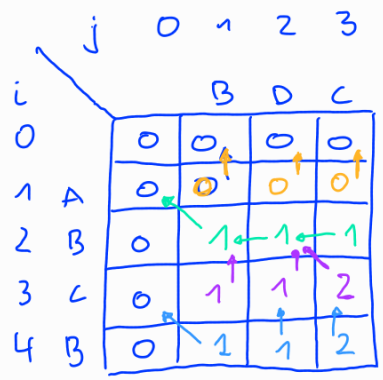
```
PRINT-LCS(b, X, i, j)   // initial call: PRINT-LCS(b, X, m, n)
    IF (i = 0 or j = 0) RETURN.
    IF (b[i,j] = "↖")
        PRINT-LCS(b, X, i-1, j-1)
        print x_i
    ELSE IF (b[i,j] = "↑")
        PRINT-LCS(b, X, i-1, j)
    ELSE
        PRINT-LCS(b, X, i, j-1)
```

RUNTIME: $\Theta(m+n)$
since it decrements at most one of i & j in each call

Exmpl:

$X = ABCB$, $Y = BDC$

i	j	0	1	2	3
			B	D	C
0		0	0	0	0
1	A	0			
2	B	0			
3	C	0			
4	B	0			



now $i = 1..m$ / $j = 1..n$

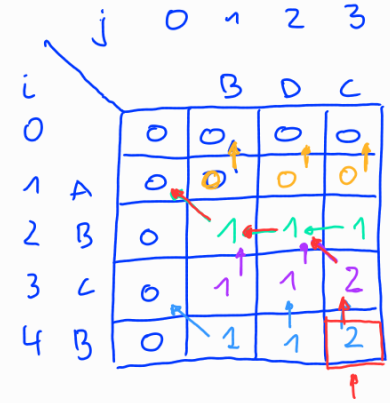
$i = 1$, $j = 1,2,3$

$i = 2$, $j = 1,2,3$

$x_2 = y_1$ "↖"

$i = 3$, $j = 1,2,3$ $x_3 = x_4$ "↖"

$i = 4$, $j = 1,2,3$ $x_4 = y_1$ "↖"



Backtracking:

start at $c[m,n]$

order "↖", "↑", "←"

but there might be further opt. solutions!

$b[3,3] =$ ↖ → print 'C'

$b[2,1] =$ ↖ → print 'B'

$\Rightarrow z = BC$

this value is length of LCS

Theorem 4.3: LCS(.) + PRINT-LCS(.) correctly returns length & LCS of given $X = x_1 \dots x_m$ & $Y = y_1 \dots y_n$ in $\Theta(mn)$ time.

proof: correctness by Thm 4.2

runtime dominated by 2 FOR-loops ($i = 1..m$, $j = 1..n$)

$\Rightarrow \Theta(mn)$

/13