Dynamic Programming

$G=(V,E)$ directed acyclic graph, each edge (u,v) has a weight $wt(u,v)>0$.

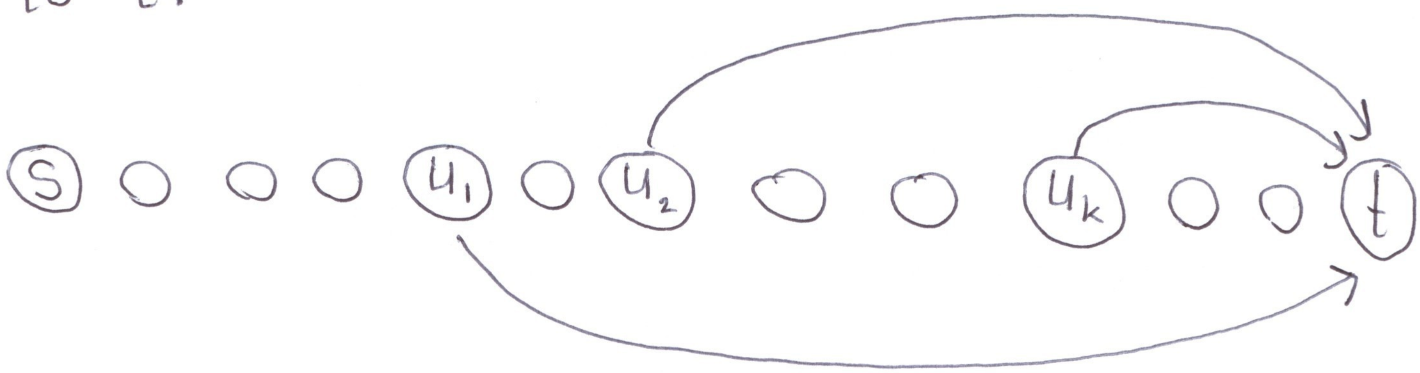
Topological sorting: vertices are numbered $v_1,\dots,v_n$ such that for each edge (v_i,v_j): $i<j$.

Let $s=v_1$, $t=v_n$.

How to compute the shortest path from s to t?

Step 1: Structure of the optimal solution

Let $u_1,u_2,\dots,u_k$ be all vertices that have an edge to t.



The last edge on the shortest path from s to t is (u_i, t) for some i, $1 \leq i \leq k$.

If we know this index i:

shortest path from s to t

$=$ $\underbrace{\text{path from } s \text{ to } u_i}$, followed by the edge (u_i, t).

this must be the shortest path from s to u_i.

But: we do not know the index i.

shortest path from s to t $=$

minimum over $i = 1, \ldots, k$ of

shortest path from s to u_i $+ \; wt(u_i, t)$

$\therefore$ shortest path from s to t contains the shortest path from s to one of $u_1, \ldots, u_k$.

Step 2: Set up a recurrence for the optimal solution

For $j = 1, 2, \ldots, n$, define

$d(v_j)$ = length of a shortest path from s to v_j.

Since $v_n = t$, we want to compute $d(v_n)$.

Recurrence:

$$d(v_1) = 0$$

for $2 \le j \le n$:

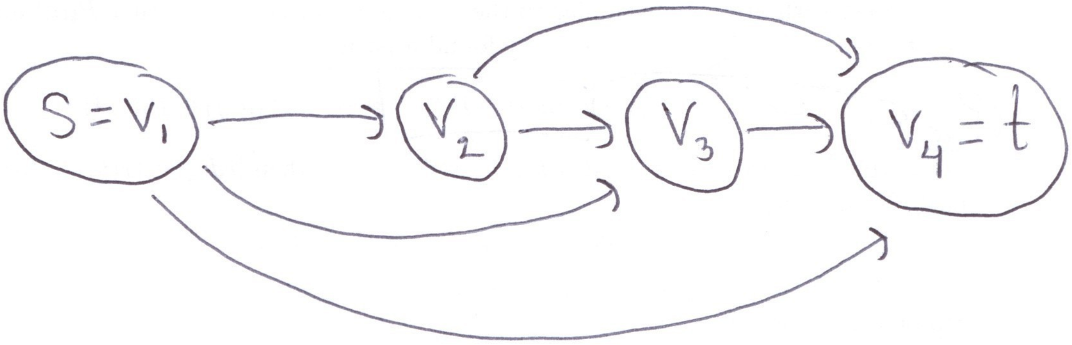
$$d(v_j) = \min\{ d(v_i) + wt(v_i, v_j) : (v_i, v_j) \text{ is an edge in } E \}.$$

Step 3: Solve the recurrence bottom-up

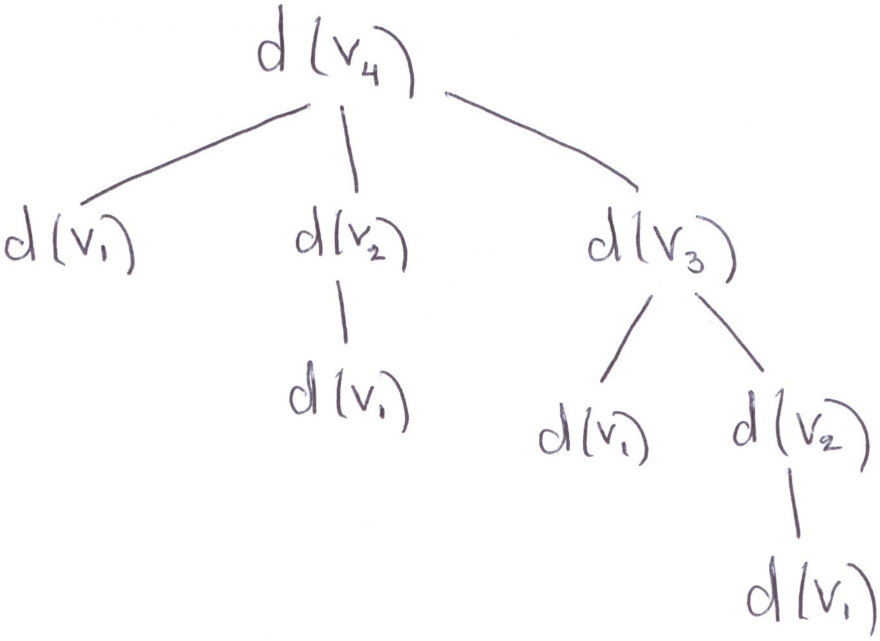
First idea: To compute $d(v_n)$, take all edges $(u_1, t), \ldots, (u_k, t)$, and recursively compute $d(u_1), \ldots, d(u_k)$. From this, compute $d(v_n)$ as

$$d(v_n) = \min\{ d(u_i) + wt(u_i, t) : 1 \le i \le k \}.$$

Example:



Recursion tree:



$d(v_1)$ is computed 4 times.

$d(v_2)$ is computed twice.

In general, this leads to an exponential running time (see pages 3-6).

Better: compute, in this order, $d(v_1), d(v_2), \ldots, d(v_n)$.

When computing $d(v_j)$, we already know $d(v_1), \ldots, d(v_{j-1})$; from these values, we obtain $d(v_j)$ without recursive calls.

Algorithm:

$d(v_1) = 0;$

for $j = 2$ to n:

  let $k = \text{indegree}(v_j)$;

  let $u_1, \ldots, u_k$ be all vertices that have an edge to v_j;

  $d(v_j) = \infty$;

  for $i = 1$ to k:

    if $d(u_i) + wt(u_i, v_j) < d(v_j)$:

      $d(v_j) = d(u_i) + wt(u_i, v_j)$;

return $d(v_n)$;

Running time

$$O\left(\sum_{j=1}^{n}\left(1+\text{indegree}(v_j)\right)\right)=O(|V|+|E|).$$

<u>Exercise</u>: We assumed that we can access the k incoming edges to v_j. If G is stored using adjacency lists, this is very slow.
Show that, in $O(|V|+|E|)$ time, we can compute for every vertex v_j, a list of all vertices that have an edge to v_j.

<u>Exercise</u>: Show that the longest path from s to t can be computed in $O(|V|+|E|)$ time.

Longest increasing subsequence

Input: sequence $a_1, \dots, a_n$ of numbers.

if $1 \leq i_1 < i_2 < \dots < i_k \leq n$, then

$a_{i_1}, a_{i_2}, \dots, a_{i_k}$ is a subsequence.

Compute LIS = length of longest increasing subsequence

$5, 2, 8, 6, 3, 6, 9, 7$ $\qquad$ LIS $= 4$

Observation:

* $\mathrm{LIS}(a_1, \dots, a_n) = \mathrm{LIS}(-\infty, a_1, \dots, a_n, \infty) - 2.$
* $\mathrm{LIS}(-\infty, a_1, \dots, a_n, \infty)$ starts with $-\infty$ and ends with ∞.

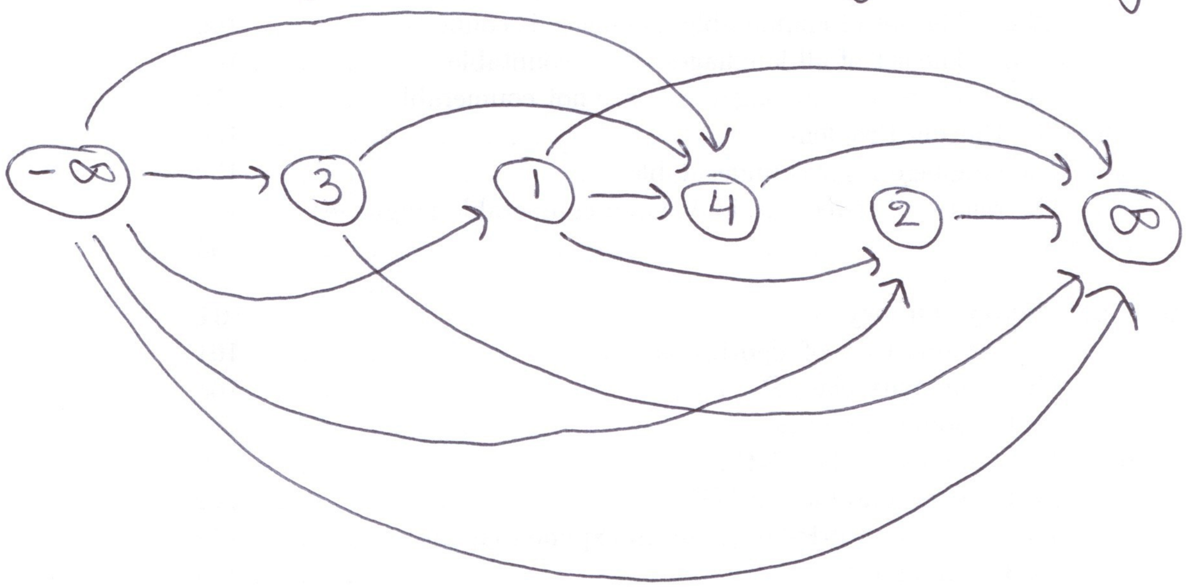
Define graph with vertices $v_0, v_1, \dots, v_{n+1}$

$key(v_0) = -\infty$, $key(v_{n+1}) = \infty$,

for $1 \leq i \leq n$: $key(v_i) = a_i$.

directed edge $(v_i, v_j) \iff i < j$ and $key(v_i) < key(v_j)$.

Example: sequence 3, 1, 4, 2 gives the graph



Give each edge weight $=1$. Then:

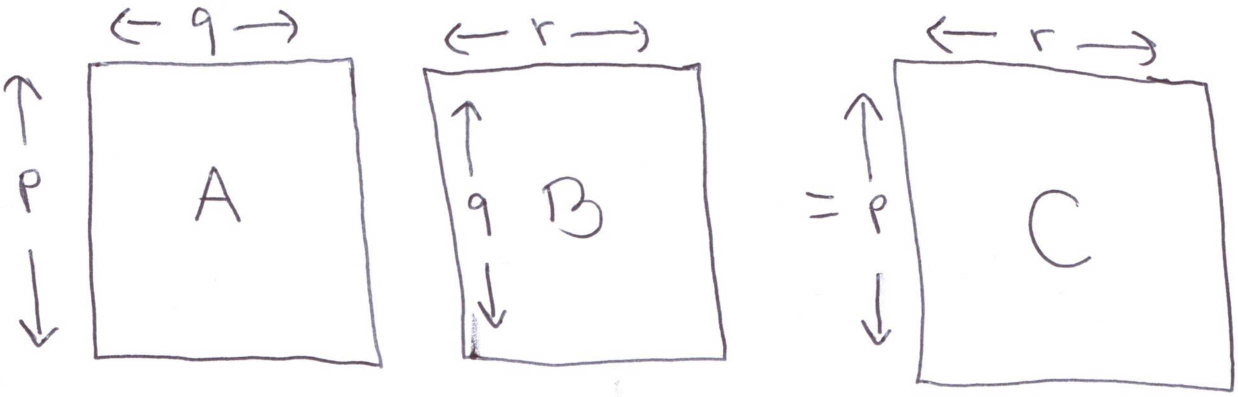
$$\mathrm{LIS}(a_1, \ldots, a_n) = (\text{length of longest path from } v_0 \text{ to } v_{n+1}) - 1.$$

We can use the previous algorithm (and the exercise on page 138).

$$\text{Running time} = O(\underbrace{|V|}_{= n+2} + \underbrace{|E|}_{\leq \binom{n+2}{2}}) = O(n^2).$$

Matrix Chain Multiplication

$$\left.\begin{array}{l} A: \ p \times q \text{ matrix} \\ B: \ q \times r \text{ matrix} \end{array}\right\} \ C = AB: \ p \times r \text{ matrix}$$



C has pr entries; each one can be computed in $O(q)$ time. $\therefore$ C can be computed in $O(pqr)$ time.

We define the <u>cost</u> of multiplying A and B to be pqr.

Consider 3 matrices

A_1 : 10×100

A_2 : 100×5

A_3 : 5×50

How to compute $A_1 A_2 A_3$:

① compute $A_1 A_2$: cost $= 10 \times 100 \times 5 = 5{,}000$

compute $\underbrace{(A_1 A_2)}_{10 \times 5} \underbrace{A_3}_{5 \times 50}$: cost $= 10 \times 5 \times 50 = 2{,}500$ +

total $= 7{,}500$

② compute $A_2 A_3$: cost $= 100 \times 5 \times 50 = 25{,}000$

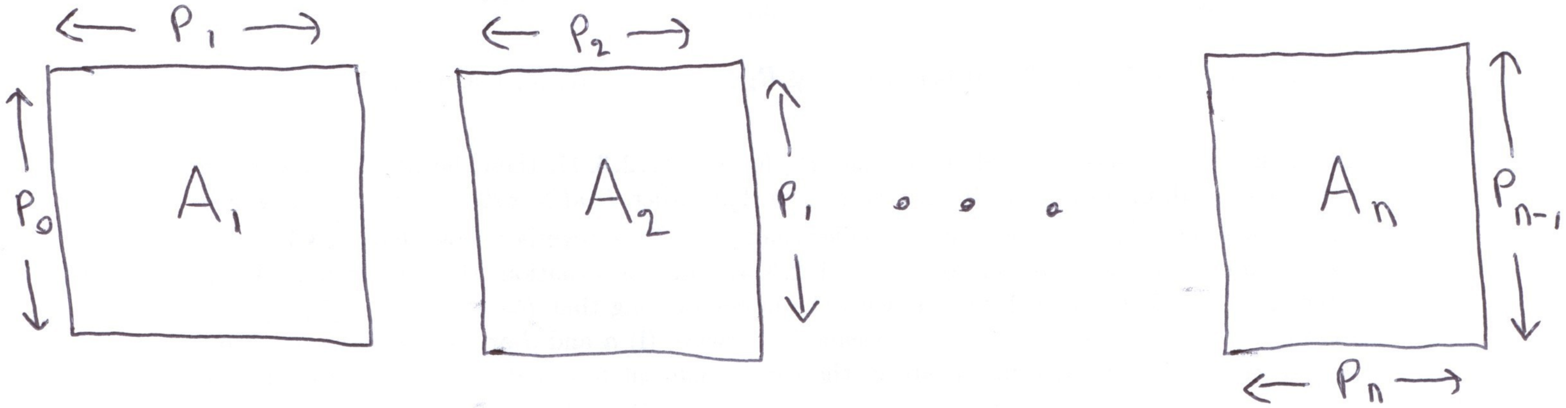
compute $\underbrace{A_1}_{10 \times 100} \underbrace{(A_2 A_3)}_{100 \times 50}$: cost $= 10 \times 100 \times 50 = 50{,}000$ +

total $= 75{,}000$

Given matrices $A_1, A_2, \ldots, A_n$,

positive integers $p_0, p_1, \ldots, p_n$,

matrix A_i has p_{i-1} rows and p_i columns



Compute the best order to compute $A_1 A_2 \cdots A_n$,

i.e., minimize the total cost.

Step 1: Structure of the optimal solution

Consider the best order to compute $A_i A_{i+1} \cdots A_j$.

In the last step, we multiply

$$\underbrace{(A_i \cdots A_k)}_{\text{already computed}} \underbrace{(A_{k+1} \cdots A_j)}_{\text{already computed}}$$

for some k, $i \le k \le j-1$.

How did we compute $A_i \cdots A_k$: in the best order.

How did we compute $A_{k+1} \cdots A_j$: in the best order.

min. cost to compute $A_i \cdots A_j$

= min. cost to compute $A_i \cdots A_k$ +

min. cost to compute $A_{k+1} \cdots A_j$ +

$p_{i-1} p_k p_j$.

Step 2: Set up a recurrence for the optimal solution

For $1 \leq i \leq j \leq n$, define

$m(i,j)$ = minimum cost to compute $A_i \cdots A_j$.

We want to compute $m(1,n)$.

If we know k, then $m(i,j) = m(i,k) + m(k+1,j) + p_{i-1} p_k p_j$.

But: we don't know k: try all values of k, $i \leq k \leq j-1$, and take the best one.

Recurrence:

for $1 \leq i \leq n$: $m(i,i) = 0$.

for $1 \leq i < j \leq n$:

$$m(i,j) = \min_{i \leq k \leq j-1} \left(m(i,k) + m(k+1,j) + p_{i-1} p_k p_j \right).$$

Step 3: Solve the recurrence bottom-up

compute, in this order,

$m(1,1), m(2,2), \ldots, m(n,n),$

$m(1,2), m(2,3), \ldots, m(n-1,n),$

$m(1,3), m(2,4), \ldots, m(n-2,n),$

$m(1,4), m(2,5), \ldots, m(n-3,n),$

$\vdots$

$m(1,n-1), m(2,n),$

$m(1,n)$

Algorithm:

for $i = 1$ to n: $m(i,i) = 0$;

for $\ell = 2$ to n:

  // compute $m(1,\ell), m(2,\ell+1), \ldots, m(n-\ell+1,n)$

  for $i = 1$ to $n-\ell+1$:

    // compute $m(i, i+\ell-1)$

    $j = i+\ell-1$;

    // compute $m(i,j)$ using the recurrence

    $m(i,j) = \infty$;

    for $k = i$ to $j-1$:

      $m(i,j) = \min\big(m(i,j),$
      $m(i,k) + m(k+1,j) +$
      $p_{i-1} p_k p_j\big)$;

return $m(1,n)$;

Running time: 3 nested loops : $O(n^3)$.

More careful counting:

$\ell : 2 \to n$

for each ℓ : $i : 1 \to n-\ell+1$

for each i : $k : i \to i+\ell-2$

Total time:

$$\sum_{\ell=2}^{n} \sum_{i=1}^{n-\ell+1} \sum_{k=i}^{i+\ell-2} 1 = \sum_{\ell=2}^{n} \sum_{i=1}^{n-\ell+1} (\ell-1)$$

$$= \sum_{\ell=2}^{n} (n-\ell+1)(\ell-1) \underset{\substack{\uparrow \\ \text{replace } \ell-1 \text{ by } \ell}}{=} \sum_{\ell=1}^{n-1} (n-\ell)\ell$$

$$= n \sum_{\ell=1}^{n-1} \ell - \sum_{\ell=1}^{n-1} \ell^2$$

$$= n\left[1+2+3+\dots+(n-1)\right] - \left[1^2+2^2+\dots+(n-1)^2\right]$$

$$= n\cdot\frac{1}{2}(n-1)n - \frac{1}{6}(n-1)n(2n-1)$$

$$= \frac{1}{2}n(n-1)\left[n - \frac{2n-1}{3}\right]$$

$$= \frac{1}{2}n(n-1)\left(\frac{1}{3}(n+1)\right)$$

$$= \frac{1}{6}n(n^2-1)$$

$$= \frac{1}{6}(n^3-n)$$

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

Longest Common Subsequence

Sequences $X = (a, b, c, b, d, a, b)$
$\qquad\qquad\qquad\qquad \uparrow\ \uparrow\ \ \ \uparrow\ \ \uparrow$
$Y = (b, d, c, a, b, a)$

$Z = (b, c, d, b)$ is a subsequence of X, but not of Y.

$LCS(X, Y)$ = longest common subsequence of X and Y:

(b, c, b, a) or (b, d, a, b)

both have length 4.

<u>Input</u>: Sequences $X = (x_1, \dots, x_m)$ and
$Y = (y_1, \dots, y_n)$.

<u>Output</u>: $Z = LCS(X, Y)$ = longest sequence that is a subsequence of X and a subsequence of Y.

Step 1: Structure of the optimal solution

$X = (x_1, \ldots, x_m)$

$Y = (y_1, \ldots, y_n)$

Consider $Z = (z_1, \ldots, z_k) = LCS(X, Y)$.

Case 1: $x_m = y_n$.

Then $z_k = x_m = y_n$ and

$$(z_1, \ldots, z_{k-1}) = LCS(x_1 \ldots x_{m-1}, y_1 \ldots y_{n-1})$$

Case 2: $x_m \neq y_n$.

Then $z_k \neq x_m$ or $z_k \neq y_n$ (or both)

Case 2a: $z_k \neq x_m$.

Then:

$$(z_1, \ldots, z_k) = LCS(x_1 \ldots x_{m-1}, y_1 \ldots y_n).$$

Case 2b: $z_k \neq y_n$.

Then:

$$(z_1, \ldots, z_k) = LCS(x_1 \ldots x_m, y_1 \ldots y_{n-1})$$

But: we do not know if we are in Case 2a or 2b.

If $x_m \neq y_n$:

$(z_1, \ldots, z_k)$ is the longer of

$LCS(x_1 \ldots x_{m-1}, y_1 \ldots y_n)$ and

$LCS(x_1 \ldots x_m, y_1 \ldots y_{n-1})$.

Step 2: Set up a recurrence for the optimal solution

For $0 \leq i \leq m$ and $0 \leq j \leq n$, define

$$c(i,j) = \text{length of } LCS(x_1 \ldots x_i, y_1 \ldots y_j).$$

We want to compute $c(m,n)$.

Recurrence:

if $i=0$ or $j=0$: $c(i,j)=0$.

if $i \geq 1$, $j \geq 1$, $x_i = y_j$:

$$c(i,j) = 1 + c(i-1, j-1).$$

if $i \geq 1$, $j \geq 1$, $x_i \neq y_j$:

$$c(i,j) = \max(c(i-1,j), c(i,j-1)).$$

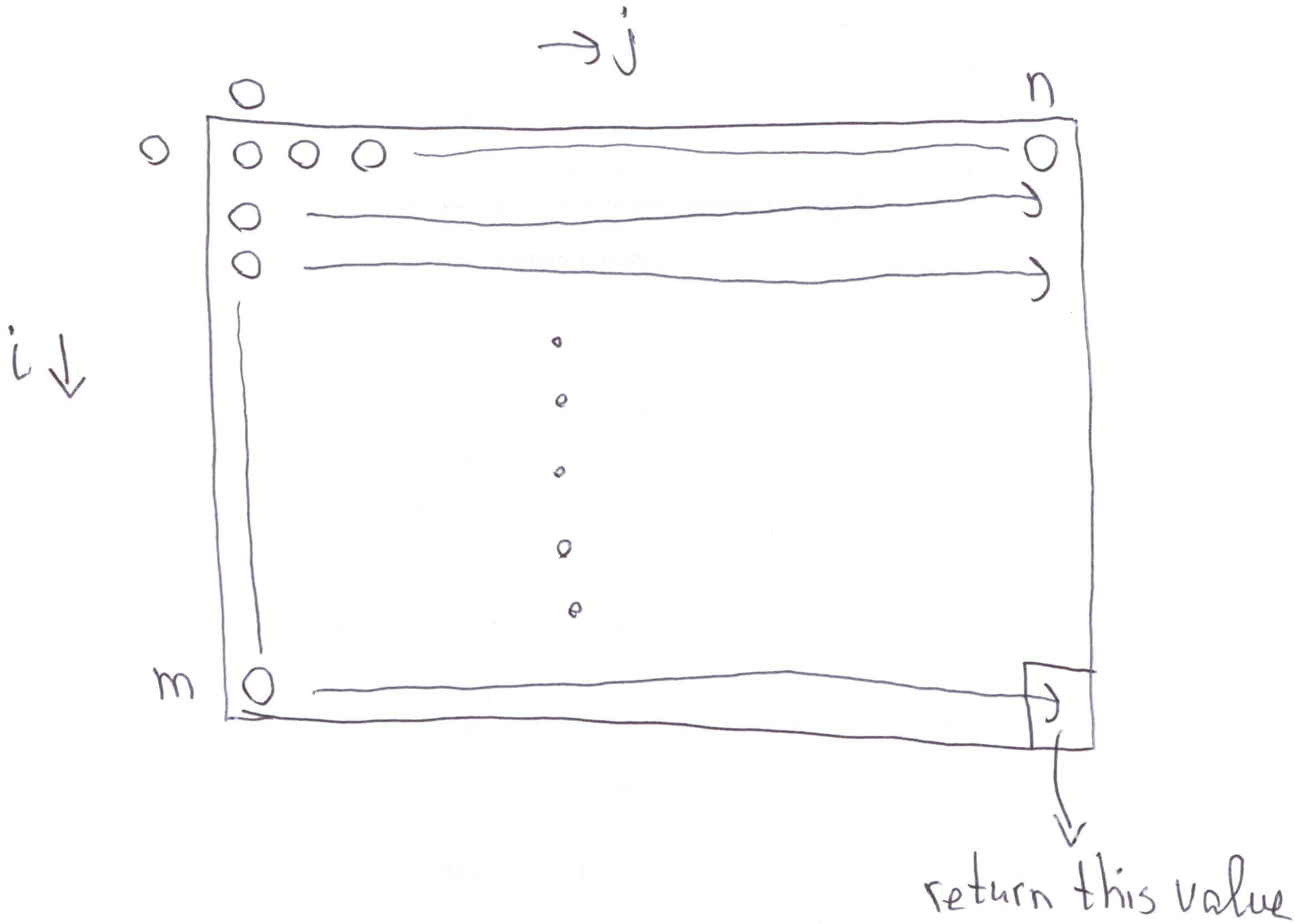
Step 3: Solve the recurrence bottom-up

Fill in the matrix $c(i,j)_{0 \leq i \leq m, 0 \leq j \leq n}$:

first row: $c(0,0) = c(0,1) = \ldots = c(0,n) = 0$

first column: $c(0,0) = c(1,0) = \ldots = c(m,0) = 0$

Then fill in the matrix, row by row; in each row from left to right:



Algorithm:

for $i = 0$ to m : $c(i,0) = 0$;

for $j = 1$ to n : $c(0,j) = 0$;

for $i = 1$ to m :

  for $j = 1$ to n :

    if $x_i = y_j$: $c(i,j) = 1 + c(i-1, j-1)$

      else : $c(i,j) = \max(c(i-1,j), c(i,j-1))$;

return $c(m,n)$;

Running time: $O(mn)$

Space: $O(mn)$

But: we only need the current row and the previous row.

$\therefore$ space: $O(m+n)$

Exercise: Sequence $a_1, \dots, a_n$ of numbers.

Sort this sequence, denote it by $b_1 < b_2 < \dots < b_n$.

Show that

$$LIS(a_1 \dots a_n) = LCS(a_1 \dots a_n, b_1 \dots b_n).$$

Dynamic Programming:

Step 1: Structure of the optimal solution.

Show that there is optimal substructure:

optimal solution "contains" optimal solutions for subproblems (which are smaller problems).

Step 2: Set up a recurrence for the optimal solution.

We know from Step 1: optimal solution can be obtained from optimal solutions for subproblems.

Use this to derive recurrence relations.

Step 3: Solve the recurrence bottom-up.

first solve the smallest subproblems (usually the base case of the recurrence).

then solve the second smallest subproblems.

then solve the third smallest subproblems.

etc.

Do not use a recursive algorithm.

All-Pairs Shortest Paths

directed graph $G=(V,E)$, where $V=\{1,2,\ldots,n\}$.

each edge (i,j) has a weight $wt(i,j)>0$.

for all i and j, compute $\delta_G(i,j)$ = weight of a shortest path from i to j.

Step 1: Structure of the optimal solution.

Consider the shortest path from i to j, andasssume this path has at least one interior vertex.

Let k be the largest interior vertex.

shortest path from i to k; all interior vertices are $\leq k-1$

shortest path from k to j; all interior vertices are $\leq k-1$.

Step 2: Set up a recurrence for the optimal solution.

for $1 \le i \le n$, $1 \le j \le n$, $0 \le k \le n$, define

$\text{dist}(i,j,k)$ = length of a shortest path from i to j, all of whose interior vertices are $\le k$.

We want to compute

$$\text{dist}(i,j,n) = \delta_G(i,j) \text{ for all } 1 \le i \le n, 1 \le j \le n.$$

Recurrence:

for $1 \le i \le n$: $\text{dist}(i,i,0) = 0$.

for $1 \le i \le n$, $1 \le j \le n$, $i \ne j$:

$$\text{dist}(i,j,0) = \begin{cases} wt(i,j) & \text{if } (i,j) \text{ is an edge,} \\ \infty & \text{otherwise.} \end{cases}$$

for $1 \le i \le n$, $1 \le j \le n$, $1 \le k \le n$:

$$dist(i,j,k) =$$
$$\min(dist(i,j,k-1), \; dist(i,k,k-1) + dist(k,j,k-1)).$$

Step 3: Solve the recurrence bottom-up.

Algorithm: (Floyd-Warshall)

```
for i = 1 to n:
    for j = 1 to n:
        if i = j: dist(i,j,0) = 0
          else dist(i,j,0) = ∞;
for each edge (i,j): dist(i,j,0) = wt(i,j);
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
            dist(i,j,k) =
                min(dist(i,j,k-1),
                    dist(i,k,k-1) + dist(k,j,k-1));
```

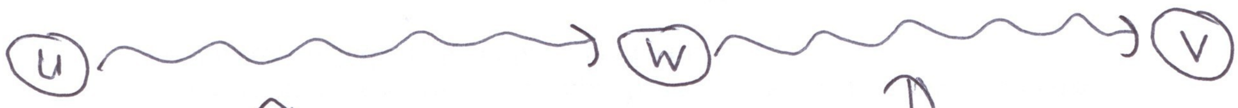
Running time : $O(n^3)$.

final remark about dynamic programming:

directed graph $G = (V, E)$, all edges have weight 1.

u, v : 2 vertices.

Assume we know a vertex w on the shortest path from u to v:



this must be a shortest path from u to w

this must be a shortest path from w to v

these 2 paths do not overlap (why?)

$\therefore$ optimal substructure.

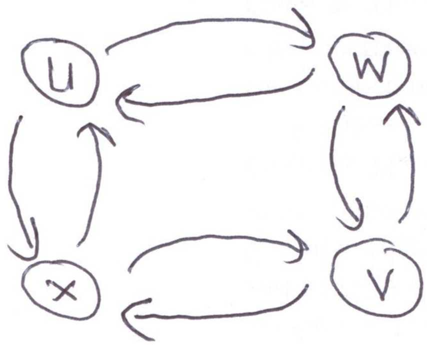
Assume we know a vertex w on the longest path from u to v:

longest path from u to w ?

longest path from w to v ?

No!

Example:



longest path from u to v:

$u \to w \to v$

not the longest path from u to w

not the longest path from w to v.

In fact: computing the longest path is NP-hard.