Today

- Finish counting path in DAG
- 2D Dyn. Prog.

Input: Directed Acyclic Graph G

Output: # paths sources $\rightsquigarrow$ sinks

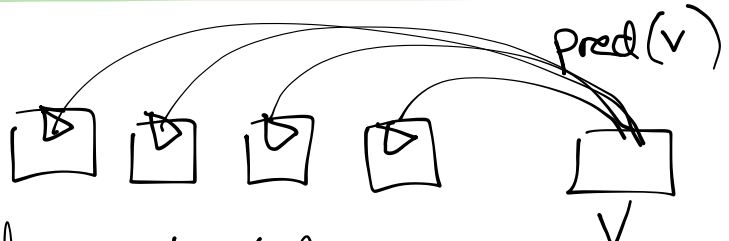
sources: no in-edges

sinks: no out-edges.

$$pred(v) = \{u \mid (u,v) \in E\}$$

Key recurrence:

$$paths(v) = \begin{cases} 1 & v \text{ source} \\ \sum_{u \in pred(v)} paths(u) & v \text{ not source} \end{cases}$$

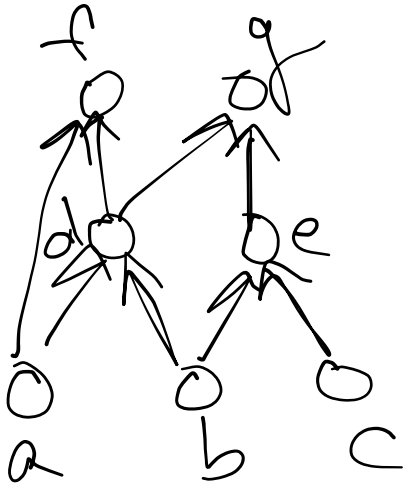


Topological sort (of a DAG)

Order vertices $v_1, \ldots, v_n$ such that if $\exists$ directed path $v_i \rightsquigarrow v_j$ then $i < j$.

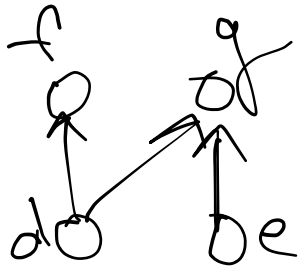
Can be found in

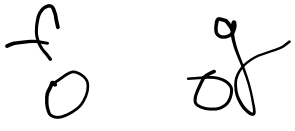
- $O(E \log V)$ by sorting
- $O(V+E)$ ← by modified DFS
  Kahn '60s "peel off sources"
  ...



a, b, c, d, e, f, g

any order | any order | any order





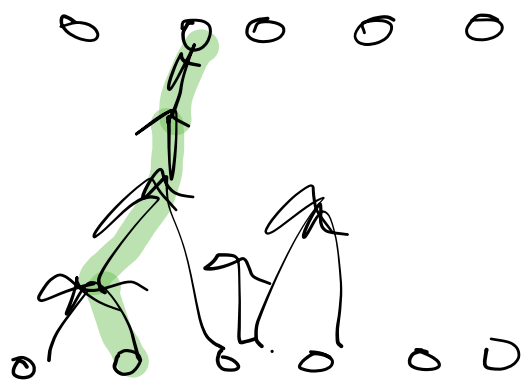
Paths in DAG(G)

```
L = TopoSort(V(G))
stillSource = True
for i=1 to |V|
    if inDeg(L[i]) = 0  //source
        paths(i) = 1
    else
        paths(i) = sum(paths(j) : j in pred(L[i]))
return paths
```

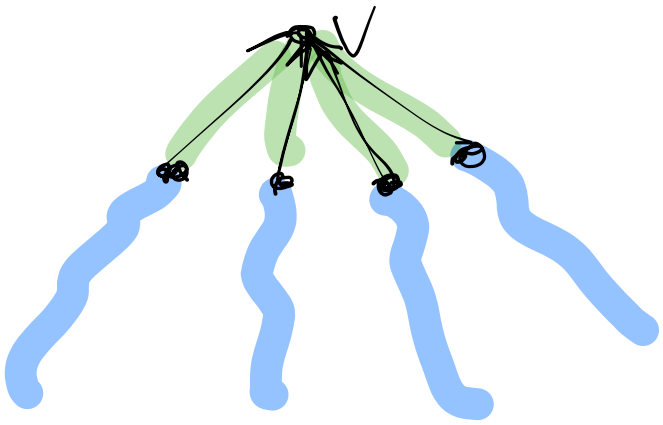
Longest Path in DAG

Input: DAG G

Output: Length of the longest directed path in G



Longest Path Ending At $(v) = \begin{cases} 0 & v \text{ source} \\ 1 + \max\{LP(u) : u \in pred(v)\} \end{cases}$



Longest Path in DAG(G)

```
L = TopoSort(V(G))

for i=1 to |V|
    if indeg(L[i])=0  //source
        LP(i)=0
    else
        LP(i)=1+max( LP(j) : j in pred(L[i]))
return LP
```

2D Dyn. Prog.

Longest Common Subsequence

Input: Two strings X, Y

A subsequence of a string X is $X_{i_1} X_{i_2} \cdots X_{i_k}$

$i_1 \leq i_2 \leq \cdots \leq i_k$

e.g. $X = ABCDE$

ACD ✓ BAD ✗

Output: Length of the longest subsequence appearing in both x and y.

e.g. $X = \underline{A}BC\underline{DE}$ $y = \underline{A}GA\underline{PE}$ → 2

e.g. $X = \underline{A}B\underline{C}\underline{DE}$ $y = \underline{A}D\underline{C}\underline{E}B$ → 3

Recurrence

Decide about last letter

Chop it off, recurse on shorter strings

$LCS(x,y)$:= length of the longest common subseq.

($n = len(x)$, $m = len(y)$)

$$LCS(x,y) = \begin{cases} 1 + LCS(x[1..n-1], y[1..m-1]) & x_n = y_m \\ \max(LCS(x, y[1..m-1]), LCS(x[1..n-1], y)) & x_n \neq y_m \end{cases}$$

Recurrence ⇝ Recursive Algo ⇝ Memoize ⇝ Figure out order in which to solve iteratively (instead of recursively)

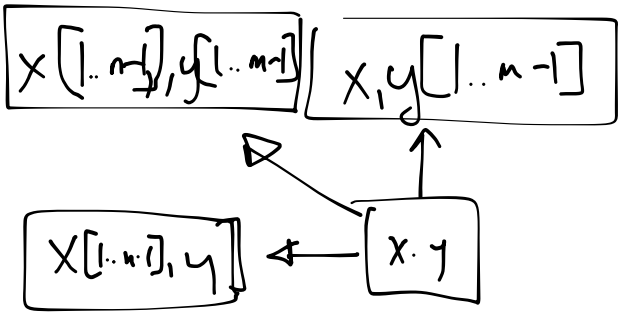
recursive

```
LCSR(x,y)
    n = len(x)
    m = len(y)
    if n = 0 or m = 0
        return 0
    else
        if x[n] = y[m]
            return 1 + LCSR(x[1...n-1], y[1...m-1])
        else
            return max(LCSR(x, y[1...m-1]),
                       LCSR(x[1..n-1], y))
```

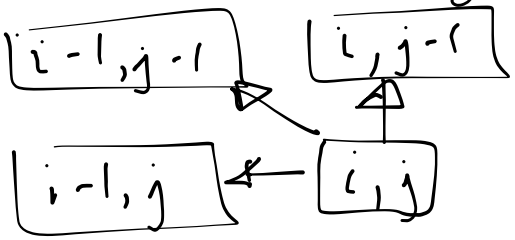
$n+m$

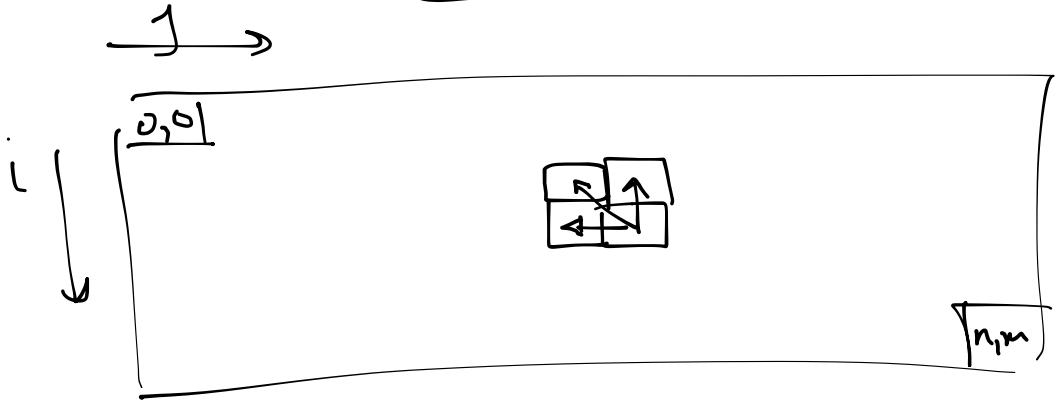
$T(l) \leq 2T(l-1) + \Theta(1) \rightsquigarrow T(l) \leq O(2^{n+m})$

Dependency Diagram



$T[i,j] := LCS(x[1 \ldots i], y[1 \ldots j])$





Fill in table top → bottom (i increasing)
left → right (j increasing)

LCS(x, y)

Runtime

T = new array [n+1][m+1] $\Theta(nm)$

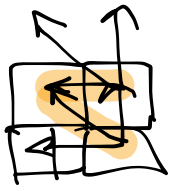
// base cases

for i = 0 to n

T[i, 0] = 0 } $\Theta(n)$

```
for j=0 to m                                   } Θ(m)
    T[0,j] = 0
// fill rest of table
for i=1 to n                                   } Θ(n)
    for j=1 to m                               } Θ(m)
        if x[i] = y[j]                         } Θ(1)
            T[i,j] = 1 + T[i-1,j-1]
        else
            T[i,j] = max(T[i-1,j], T[i,j-1])
return T[n,m].
```

$\Theta(nm)$



Runtime of recursive algo

When $x_n \neq y_m$

$$LCSR(x,y) = \max(LCSR(x[1\ldots n-1], y), LCSR(x, y[1\ldots m-1]))$$

$l = len(x) + len(y)$

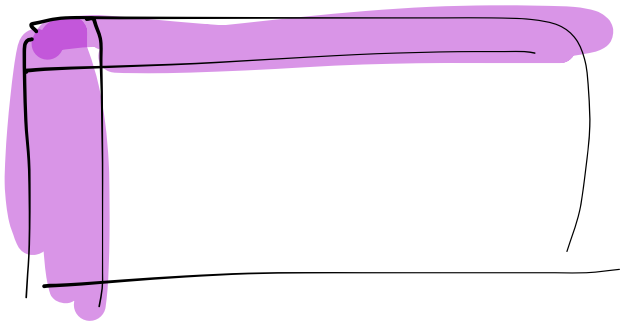
$$T(l) \leq 2T(l-1) + O(1)$$

$$\leq 4T(l-2) + \ldots$$

$$\Rightarrow O(2^l)$$

```
LCSubseq(x,y)
    T = new array[n+1][m+1]
    B = new array[n+1][m+1]   // "backpointers"

    for i = 0 to n
        T[i,0] = 0
        B[i,0] = null
    for j = 0 to m
        T[0,j] = 0
        B[0,j] = null
    for i = 1 to n
        for j = 1 to n
            if x[i] = y[j]
                T[i,j] = 1 + T[i-1,j-1]
                B[i,j] = (i-1, j-1)
            else
                if T[i-1,j] > T[i,j-1]
                    T[i,j] = T[i-1,j]
                    B[i,j] = (i-1, j)    // T[i-1,j] = max
                else
                    T[i,j] = T[i,j-1]
                    B[i,j] = (i, j-1)
    // Backtrace thru B
    subseq = ""
    cur = (n,m)
```



```
while B[cur] ≠ null
    if B[cur] = (cur_1 - 1, cur_2 - 1)
        subseq.append(x[cur_1])
    else
        cur = B[cur]
return subseq.reverse()
```

LCS

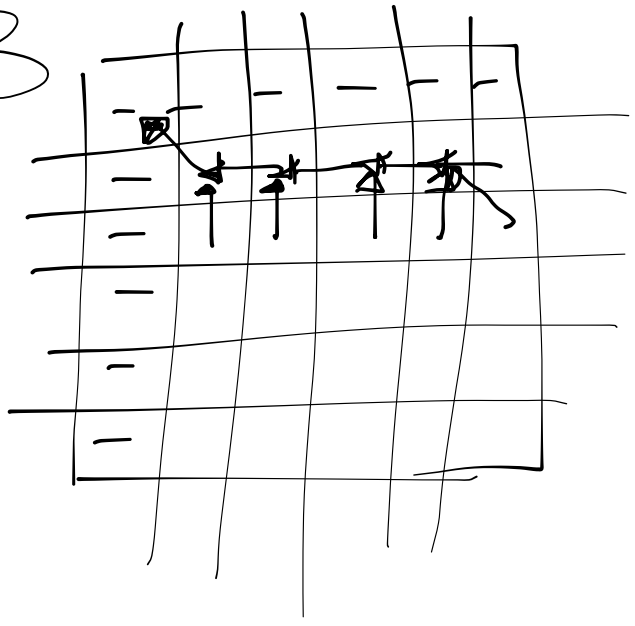
$X = \underline{A}\,B\,\underline{C}\,D\,\underline{E}$

$Y = \underline{A}\,D\,\underline{C}\,\underline{E}\,B$

T

	-	A	D	C	E	B
-	0	0	0	0	0	0
A	0	1	1	1	1	1
B	0	1	1	1	1	2
C	0					
D	0					
E	0					

B



Today:

- Finish Longest Common Subsequence
- More 2D examples
  - Knapsack
  - String Alignment / Edit Distance

Longest Common Subsequence

Input: Two strings X, Y

A subsequence of a string X is $X_{i_1} X_{i_2} \cdots X_{i_k}$

$i_1 \le i_2 \le \cdots \le i_k$

e.g. $X = ABCDE$

ACD ✓ BAD ✗

Output: Length of the longest subsequence appearing in both x and y.

e.g. $X = \underline{A}BC\underline{DE}$ $y = \underline{A}GA\underline{PE}$ → 2

e.g. $X = \underline{A}B\underline{C}\underline{DE}$ $y = \underline{A}D\underline{C}\underline{E}B$ → 3

Recurrence

Decide about last letter

Chop it off, recurse on shorter strings

$LCS(x,y)$:= length of the longest common subseq.

$n = len(x)$ $m = len(y)$

$$LCS(x,y) = \begin{cases} 1 + LCS(x[1..n-1], y[1..m-1]) & x_n = y_m \\ \max(LCS(x, y[1..m-1]), \\ \quad LCS(x[1..n-1], y)) & x_n \neq y_m \end{cases}$$

eg

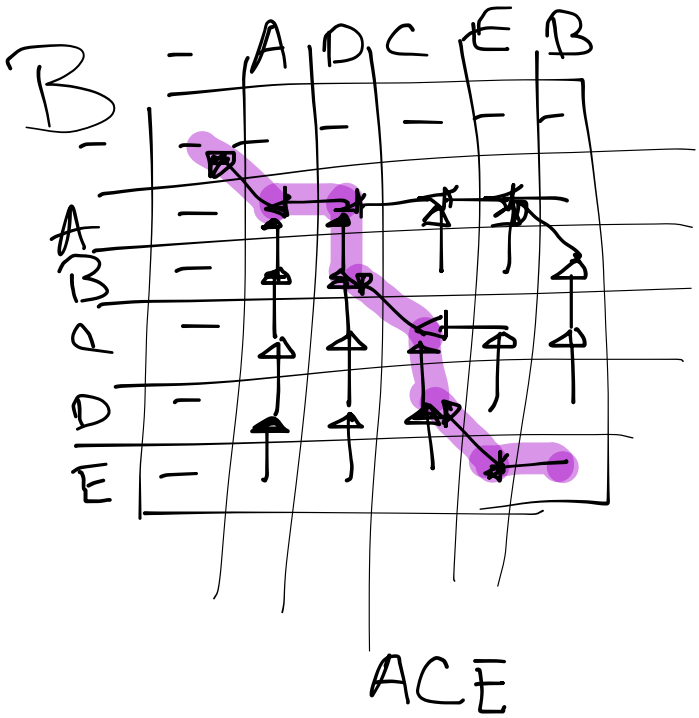
$X = A\,B\,C\,D\,E$

$y = A\,D\,C\,E\,B$

T

	-	A	D	C	E	B
-	0	0	0	0	0	0
A	0	1	1	1	1	1
B	0	1	1	1	1	2
C	0	1	1	2	2	2
D	0	1	1	2	2	2
E	0	1	1	2	3	3

B



ACE

Knapsack

Input: $[(v_1, w_1), \ldots, (v_n, w_n)], W$

Assume all $v_i, w_i, W \in \mathbb{N}$

v_1: value of item 1; w_1: weight of item 1; v_n: val of item n; w_n: weight "

Output: $S \subseteq \{1, \ldots, n\}$ such that

(1) $\sum_{i \in S} w_i \leq W$

(2) maximize $\sum_{i \in S} v_i$ (subject to (1))

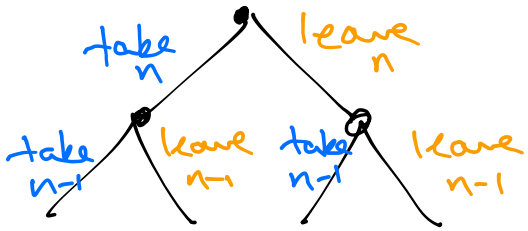
e.g.

$[(10, 10), (7, 5), (7, 5)]$ $W = 10$

```
RecKnapsack([(v_1,w_1),...,(v_n,w_n)], W)
                      L
  if n=0
    return
  else
    if w_n > W
      return RecKnapsack(L[1..n-1], W)
    else
      return max(v_n + RecKnapsack(L[1..n-1], W-w_n),
                 RecKnapsack(L[1..n-1], W))
```

Take n

Leave n



n

$\rightsquigarrow$ runtime $\Theta(2^n)$

$T[i,w] \stackrel{\text{def}}{=}$ Knapsack$(L[1 \ldots i], w)$

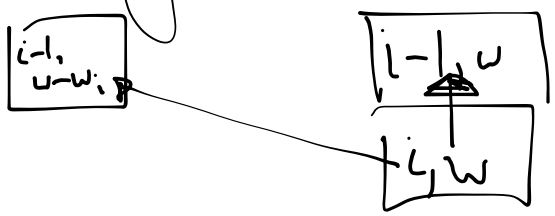
$$T[i,w] = \begin{cases} T[i-1,w] & \text{if } w_i > w \\ \max(v_i + T[i-1, w-w_i],\ T[i-1,w]) & \text{if } w_i \le w \end{cases}$$

Want $T[n,W]$

n: whole list

W: original weight threshold

Dependency Diagram



⇝ Fill in raw i-1 before raw i

⇝ Fill the table raw by raw, start w/ raw 0 up to raw n.

e.g.

$L = [(10,2), (7,1), (7,1)]$, $W = 2$

T

i \ w	0	1	2
0	0	0	0
(10,2) 1	0	0	10
(7,1) 2	0	7	10
(7,1) 3	0	7	14

B

	0	1	2
0			—
1			
2			
3			

item 3
item 2
(not item 1)

Filling in one entry
⇝ $\Theta(1)$ steps

⇒ runtime $= \Theta$(size of table)
$= \Theta(nW)$

```
Knapsack(L, W)
    n = len(L)
    T = new array[n+1][W+1]   //ints
    B = new array[n+1][W+1]   //"pointers"
    // base cases
    for w = 0 to W
        T[0,w] = 0
        B[0,w] = null

    for i = 1 to n
        for w ∈ [0..W]
            if w_i > w
                T[i,w] = T[i-1,w]
                B[i,w] = (i-1, w)
            else
                take = v_i + T[i-1, w-w_i]
                leave = T[i-1, w]
```

```
if take > leave
    T[i,w] = take
    B[i,w] = (i-1, w-w_i)
else
    T[i,w] = leave
    B[i,w] = (i-1, w)

// Backtrace to get soln
out = []
cur = (n, W)
while B[cur] ≠ null
    if B[cur] = (cur_1 - 1, cur_2)
        continue
    else
        out.append(cur_1)
return out
```

weight threshold unchanged

"i^{th} item"

Dynamic Prog. Outline

Recursive algo (that tries all possibilities)

Memoize
→ Recognize repeated "sub-problems"/recursive calls
Recurrence + dependency diagram
⟹ order to fill in table

Iterative table-based algo
Backtracing to find solution

Extra Optimizations

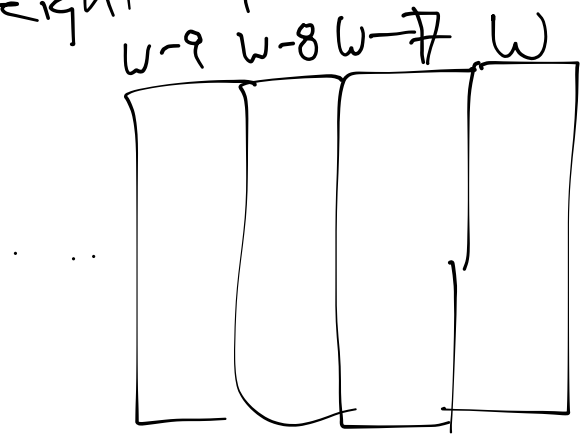
Do we need all W cols?

e.g. $[(10,10), (7,5), (7,5)]$ $W = 10$

note: the only w that will occur are: 10, 5, 0

	0	5	10
0			
1			
2			
3			

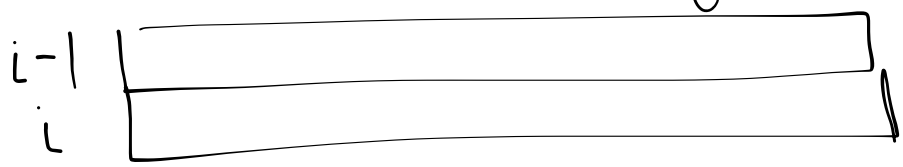
e.g. if min weight = 7



Saving Space

If you just want <u>value</u> of the opt (no backtracing)

Then — only need to keep two rows in memory @ a time



⇝ space $\Theta(W)$ vs. whole table $\Theta(nW)$

Dyn. prog. {Memoization?} is a way to trade space for time.

	Brute force	Dyn Prog.
time	$\Theta(2^n)$	$\Theta(nW)$
space	$\Theta(n)$ ↑ keep track of call stack = Θ(height of call tree)	$\Theta(nW)$ (or $\Theta(W)$ w/ trick)