Lecture 12 – Dynamic Programming

Recap

Main Idea: To solve a big problem:

- Identify smaller subproblem s.t. a solution to the big problem can be derived from solutions to subproblems.
- Solve all subproblems from "small to large"
- Analyze runtime & Memory.

Alternative View: Recursion with Memoization
(avoids doing the same subproblem again & again).

Example: "Big" problem: Given n calculate Fib_n

Subproblems: for $i = 2, 3, \ldots, n$ calculate Fib_i

Code:

$F_0 = 0$, $F_1 = 1$

for $i = 2, \ldots, n$

$F_i = F_{i-1} + F_{i-2}$

Recursion w. Memoization

```
def fibMem(n):
    if n ≤ 1: return n
    if n in Mem: return Mem[n]
    Mem[n] = fibMem(n-1) + fibMem(n-2)
    return Mem[n]
```

More examples from last time:

Shortest path in a DAG
Longest path in a DAG.

Problem 2: Longest Increasing Subsequence (LIS)

- **Input:** Array of n numbers, e.g. $x_1, x_2, \dots, x_n$

  1, 3, 2, 7, 4, 5, 6

- **Goal:** Find longest subsequence (non-consecutive) that is strictly increasing.

Greedy: not optimal. optimal: 1, 2, 4, 5, 6
or 1, 3, 4, 5, 6

Subproblems: First try $\forall i = 1, \dots, n$: $f(i)$ = Longest increasing subsequence in $x_1, \dots, x_i$

$f(n)$ from $f(1), \dots, f(n-1)$???

Second Try $\forall i = 1 \dots n$ $f(i)$ = LIS in $x_1, \dots, x_i$ that includes x_i.

$$f(n) = \max\left(1, \max_{\substack{i<n \\ x_i < x_n}} \left(f(i)+1\right)\right)$$

the longest subsequence that ends in x_i

n subproblems

Each can be solved from prev ones with additional time $O(n)$.

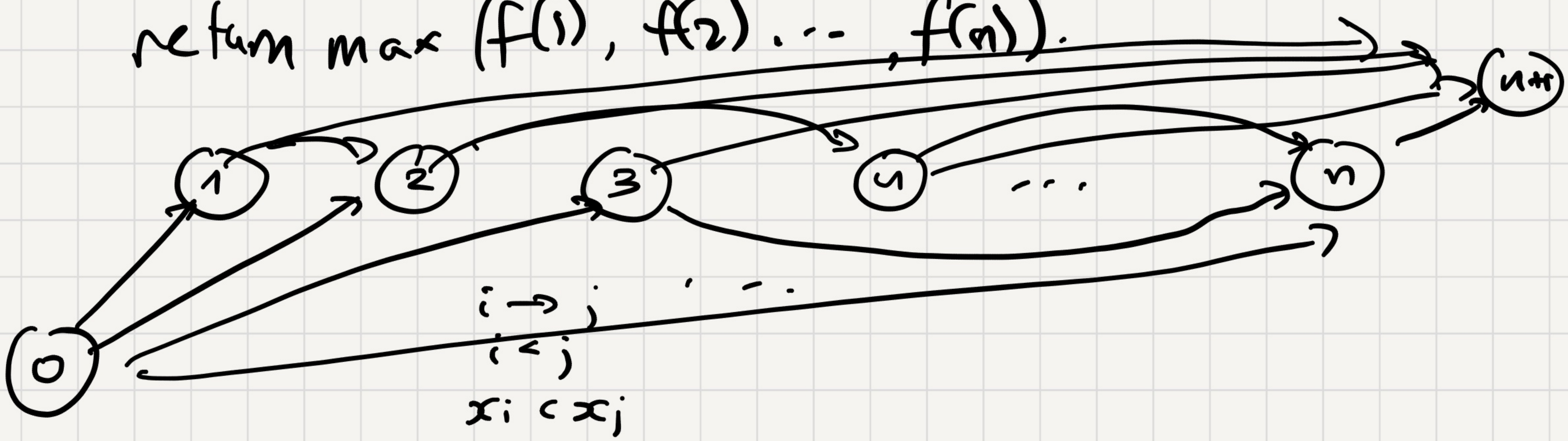
$$f(i) = \max\left(1, \max_{\substack{j<i \\ x_j < x_i}} (1 + f(j))\right)$$

<u>Runtime</u>: $O(n^2)$ time.

<u>Memory</u>: $O(n)$ memory.

Subproblems: $f(i)$ = longest subseq in $x_1, \dots, x_i$ that uses x_i.

return $\max(f(1), f(2), \dots, f(n))$.



Problem 3: Edit Distance

(Levenshtein Distance)

Given Two Strings: $S[1\ldots n]$ $T[1\ldots m]$

Find fewest number of edits to turn S into T.

Edits allowed:
1. Insert character to S.
2. Delete " from S.
3. Substitute a character for another.

Example: $S =$ "snowy"
$T =$ "sunny"

s n o w y
s $\boxed{u}$ n o w y — insert u
s u n $\boxed{n}$ w y — replace o → n
s u n n ~~w~~ y — delete w

s _ n o w y
s u n n _ y

cost of this alignment 3.

s _ n o w y _
_ s _ u n n y

DP: Subproblems:

$\Rightarrow$ • For $0 \le i \le n$ $\quad 0 \le j \le m$ $\quad f(i,j) = \text{EditDistance}(S[1\ldots i], T[1\ldots j])$

• $S[1\ldots i]$ $\quad T[1\ldots j]$

look at last char in optimal alignment

$$\begin{array}{ccc} S[i] & - & S[i] \\ - & T[j] & T[j] \\ 1 + f(i-1,j) & 1 + f(i,j-1) & f(i-1,j-1) + \delta_{i,j} \end{array}$$

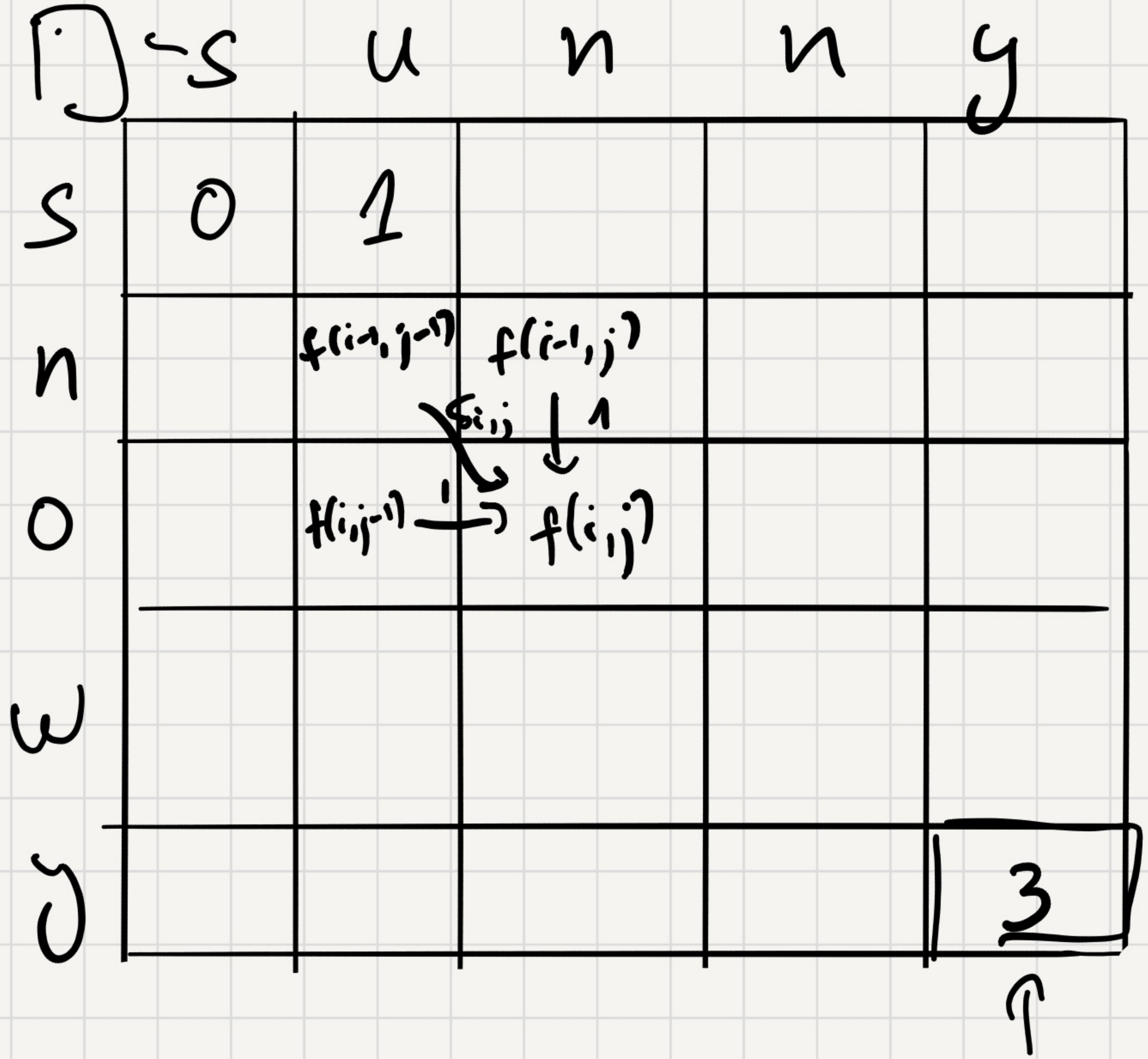
$$\delta_{i,j} = \begin{cases} 0 & S[i] = T[j] \\ 1 & \text{o.w.} \end{cases}$$

$$\boxed{f(i,j) = \min\left(1 + f(i-1,j),\ 1 + f(i,j-1),\ f(i-1,j-1) + \delta_{i,j}\right)}$$

• Edge Cases: $f(i,0) = i$ $\quad f(0,j) = j$

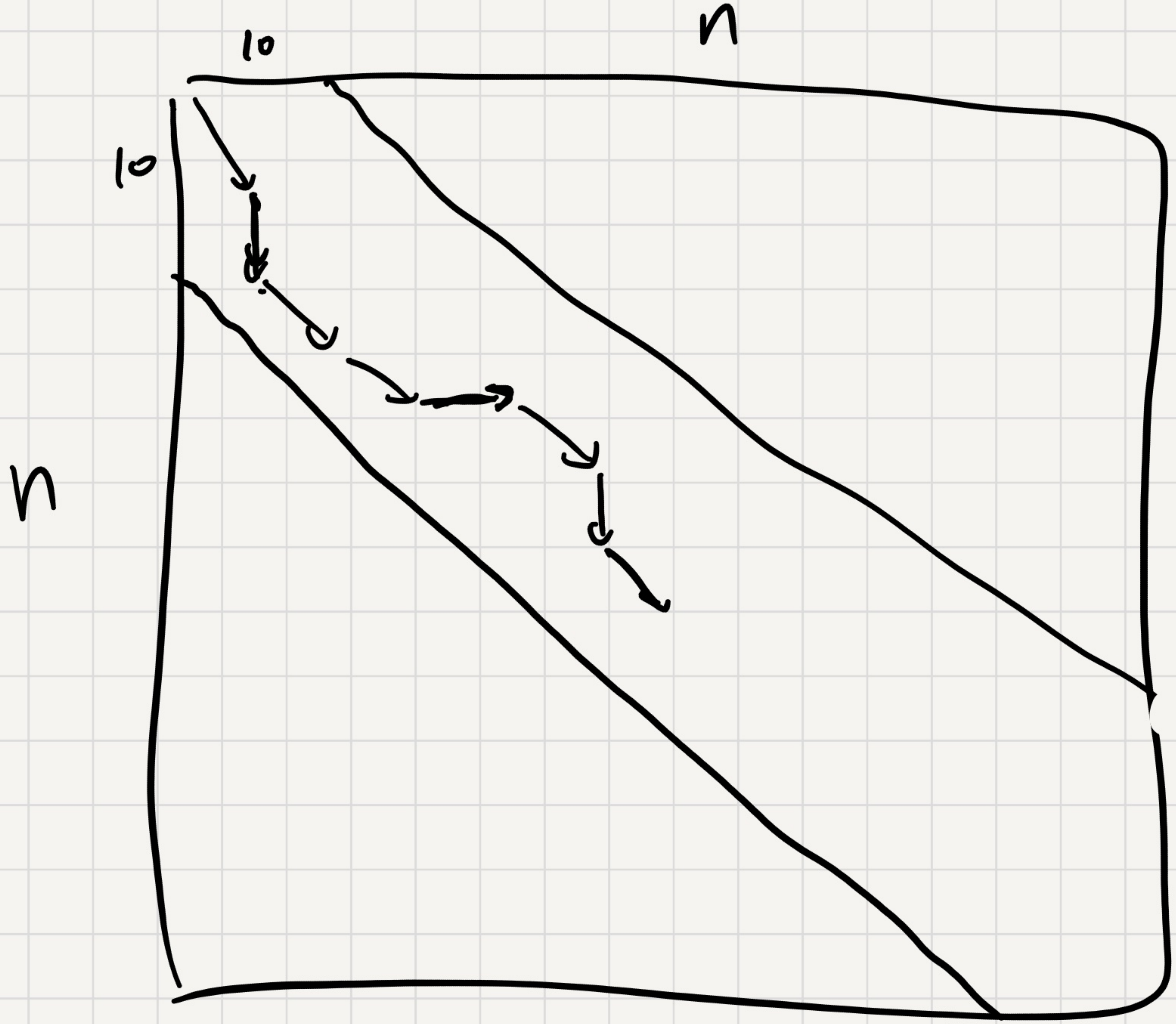
• Runtime: $(n+1)(m+1)$ Subproblems, $O(1)$ to compute $\Big\}\ O(m \cdot n)$ time.

• Memory: $O(nm)$.



Mem: $O(m)$
row by row

$O(n)$
col by col



Edit distance ≤ 10

$O(10 \cdot n)$

Example 4: Knapsack.

We're robbing a bank!

We find n items in the safe with weights $w_1, \dots, w_n$ & values $v_1, \dots, v_n$. integers

We have a bag (knapsack) that can carry at most W pounds.

Goal: Find most valuable choice that fits the knapsack.

Example:

$w_1 = 11 \quad v_1 = 15$

$w_2 = 10 \quad v_2 = 10$

$w_3 = 10 \quad v_3 = 10$

$\Bigg\} \; W = 20$

Greedy: Pick every time the item that max $\frac{v_i}{w_i}$

not work.

DP: $f(i, u)$ = max value when packing items $1, \dots, i$ in a bag of capacity u. $\Big\}$ $n \cdot W$ subproblems

$$f(i,u) = \max\big(f(i-1, u), \; \underbrace{f(i-1, u-w_i)}_{} + v_i \big)$$

if $w_i \le u$.

$$f(i,u) = \begin{cases} f(i-1,u), & \text{if } w_i > u \\ \max(f(i-1,u), f(i-1,u-w_i) + v_i) & \text{o.w.} \end{cases}$$

Runtime: $O(n \cdot W)$. Is this a polynomial-time algorithm? No!

A polynomial time alg. runs in time polynomial in the input length.

Here, the input is:

$(v_1, \ldots, v_n)$, $(w_1, \ldots, w_n)$, W

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

so runtime can be exp. in input length.

For example, if $W = 2^n$.

Runtime: $O(n \cdot 2^n)$ **Input Length**: $O(n^2)$ bits.

---

* This algorithm would be polynomial time we further assume that W is at most polynomial in n.

Example 5: Shortest Path in General Graphs

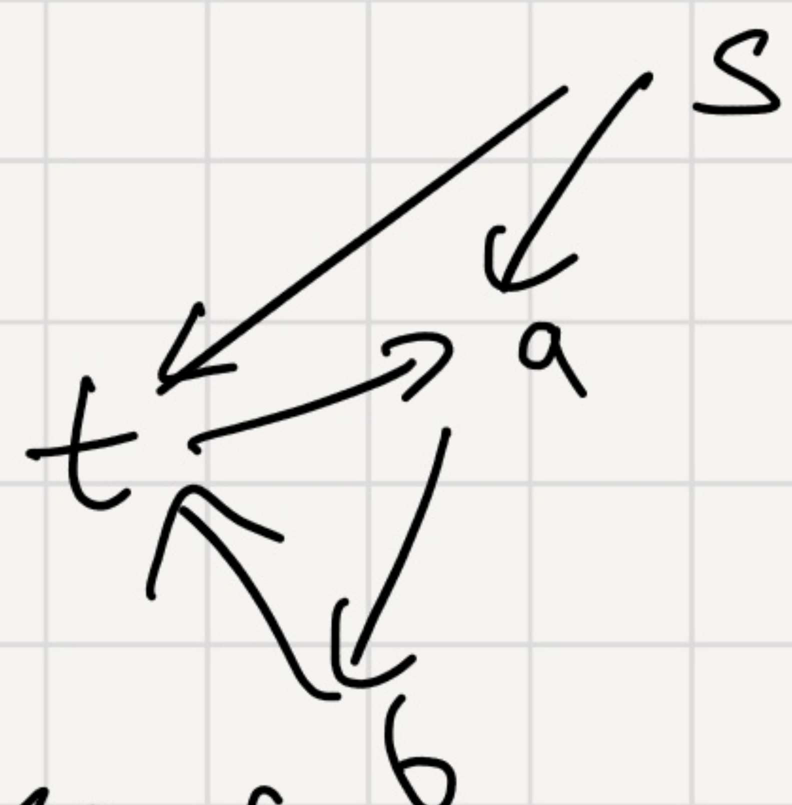
$G=(V,E)$ $\quad w: E \to \mathbb{Z}$ (either positive or negative)

Assume: no negative cycles.

Given s, t

<u>Goal</u>: comp. distance from s to t.

<u>Subproblems:</u> distance from s to v for any $v \in V$.



Doesn't work.

<u>Subproblems:</u> $dist(v,i)$ = Distance from s to v with at most i edges

$$dist(v,i) = \min\left(dist(v,i-1), \min_{u:(u,v)\in E} \left(dist(u,i-1) + \ell(u,v)\right)\right)$$

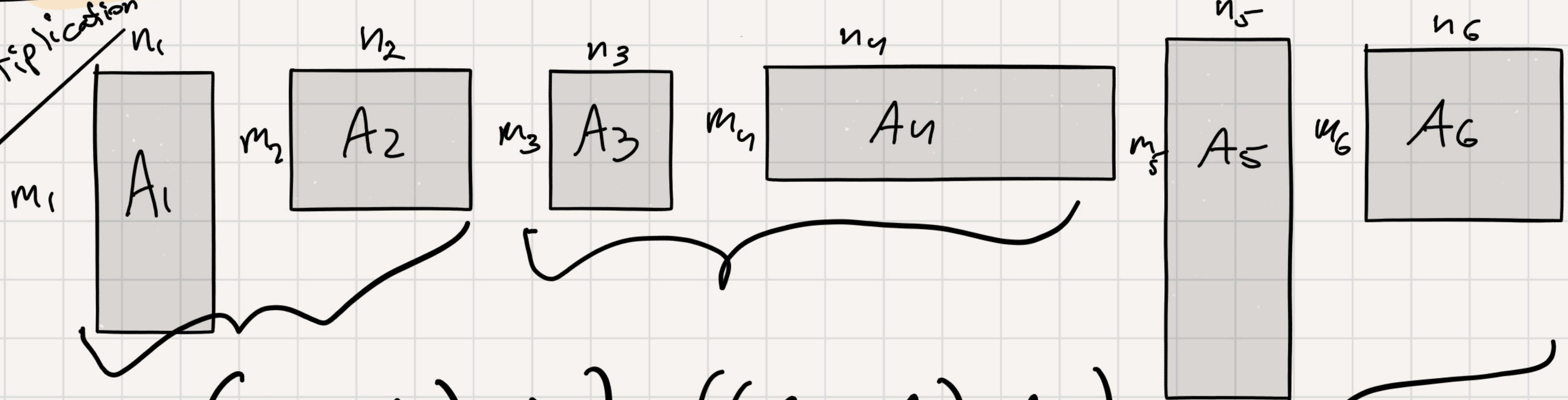
<u>Runtime:</u> # subproblems : n choices for i $\times$ n choises for v.

For subproblem (v,i) : time $= O(1+indeg(v))$.

$$\sum_{i=1}^{n} \sum_{v\in V} O(1+indeg(v)) = n \cdot O(n+m).$$

More Great Problem in the Book

Chain Matrix Multiplication



$$((A_1 \cdot A_2) \cdot A_3) \cdot ((A_4 \cdot A_5) \cdot A_6)$$

APSP : $O(n^3)$ algorithm.

TSP: 2^n time alg. $\Rightarrow$ Better than brute-force $O(n!)$ time.

Independent sets in Trees ...