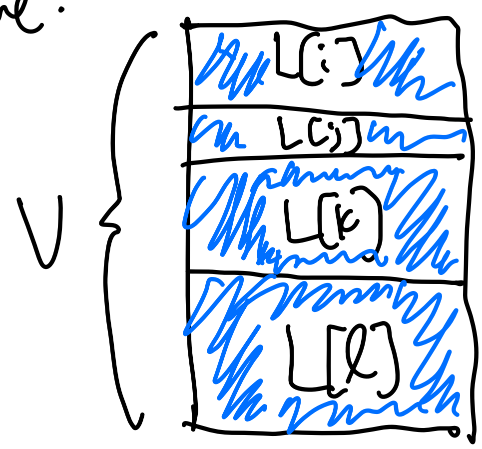
CS 331, Fall 2025
Lecture 6 (9/15)

Today:
- Knapsack
- LCS & friends
- Palindromes
- Game theory

Knapsack (Part III, Section 3.3)

Recall subset sum from last time:



Input: L: list of n natural #'s

$V \in \mathbb{N}$: target value

Output: True/False, $\exists S \subseteq [n]$, $\sum_{i \in [n]} L[i] = V$?

Key idea: 2D memoization, prefix x value

General strategy applies to integer-constrained optimization problems, e.g. knapsack

Input: W: n natural #'s

$B \in \mathbb{N}$: target value

Output: $\exists S \subseteq [n], \sum_{i \in S} W[i] = B$?

Subset sum

Input: W: n natural #'s (weights)

$B \in \mathbb{N}$: weight budget

V: n real positive #'s (values)

0-1 knapsack

Output: $\max_{S \subseteq [n]} \sum_{i \in S} V[i]$

$\sum_{i \in S} W[i] \leq B$

Applications: budget-constrained decision making problems.

Before: $S[j][b]$ = Can you hit target b using first j items?
$j \in [n], b \in [B]$

Intuition: decision to take/not take $L[j]$ affects remaining <u>value</u>, must store

Now: $S[j][b]$ = max value w/ weight budget b using first j items
$j \in [n], b \in [B]$

DP formula:

$$S[j][b] = S[j-1][b] \text{ OR } S[j-1][b-W[j]]$$

Subset sum

$$S[j][b] = \max\left(S[j-1][b],\ V[j] + S[j-1][b-W[j]] \right)$$

0-1 knapsack

Also runs in $O(nB)$ time. (row-by-row)

$\boxed{\text{Extension}}$ Unbounded knapsack

You can take multiple copies of any item.

Goal: maximize $\sum_{i \in [n]} c_i V[i]$ s.t.

$$c \in \mathbb{Z}_{\geq 0}^{n} \quad \text{(counts)}$$

$$\sum_{i \in [n]} c_i W[i] \leq B$$

DP: $S[b]$ = max achievable w/ budget b

$$S[b] = \max\Big(\underset{\text{take nothing}}{0}, \max_{\substack{i \in [n] \\ W[i] \leq b}} \underbrace{S[b - W[i]] + V[i]}_{\text{take item } i}\Big)$$

Runtime: $\underset{\#\text{ subprobs}}{O(B)} \times \underset{\text{time/subprob}}{O(n)} = O(nB)$

Longest Common Subsequence (Part III, Section 4.1)

Strings Ω = universe

e.g. $\{\underbrace{\text{'a'}, \text{'b'}, \ldots, \text{'z'}, \text{'1'}, \ldots, \text{'0'}}_{\text{characters}}\}$

String of length n: ordered list of n characters from Ω

"algorithms" $\equiv \{\text{'a'}, \text{'l'}, \ldots \text{'m'}, \text{'s'}\}$

Substring: contiguous sublist "algo"

Subsequence: delete any characters, concatenate the rest "arms"

LCS: natural distance measure on strings

Input: X, length - m string
Y, length - n string
(e.g. DNA)

Output: $|Z|$, largest possible length of common subsequence Z of X & Y

$\boxed{\text{Example}}$

X = "algorithms"

Y = "complexity"

$LCS(X, Y) = 3$

Conclusion: they are both $\underbrace{\text{"lit"}}_{\arg\max_Z |Z|}$

Key idea: 2-D DP again

$S[i][j] = LCS(X[:i], Y[:j])$

prefixes

2 cases: Can we match $X[i]$, $Y[j]$?

Case 1: No $(X[i] \neq Y[j])$

e.g. "algor", "compl" $(i=5, j=5)$

What is last char of Z?

$X[i]$, $Y[j]$, or neither. (not both)

If not $X[i]$, Plan A: $S[i-1][j]$

If not $Y[j]$, Plan B: $S[i][j-1]$

Case 2: Yes ($X[i] = Y[j]$)

e.g. "al", "compl" ($i=2$, $j=5$)

Now we can make $X[i] = Y[j]$ last character in Z.

Plan C: $1 + S[i-1][j-1]$

Summary:

$$S[i][j] = \max\begin{cases} S[i-1][j], & \text{no } X[i] \\ S[i][j-1], & \text{no } Y[j] \\ S[i-1][j-1] + \mathbb{1}(X[i] = Y[j]) & \text{both} \end{cases}$$

Runtime: $O(mn)$

$\boxed{\text{Extension}}$ Multiple strings

$LCS(W, X, Y)$: longest mutually common subsequence

lengths ℓ m n

Same idea.

$$S[i][j][k] = LCS(W[:i], X[:j], Y[:k])$$

$$= \max\big(S[i-1][j][k],\ S[i][j-1][k],\ S[i][j][k-1],\ S[i-1][j-1][k-1] + \mathbb{1}(W[i]=X[j]=Y[k])\big)$$

$\boxed{\text{Extension}}$ Edit distance

How many ops needed to turn X to Y?

Ops: Insert, Delete, Substitute

Example

X = "kitten"

Y = "sitting"

3 steps: "kitten" $\to$ "kittin" $\to$ "sittin" $\to$ "sitting"

Observation: Suppose no substitutions.

optimal edit sequence:

$$X \xrightarrow[\text{(deletions)}]{} Z$$

$$Z \xrightarrow[\text{(insertions)}]{} Y$$

Z is LCS of X & Y

Proof: 1) All deletions from X in optimal moves.

2) Rearrange so all deletions first.

3) $X \to Z \to Y$ shortest if Z longest.

Edit distance: Same idea.

All moves are 1) Delete from X

2) Insert from Y

3) Substitute X to Y

or sequence can be made shorter.

$S[i][j]$ = Edit distance of $X[:i]$, $Y[:j]$

$$S[i][j] = \min\begin{pmatrix} S[i][j-1]+1 & \text{insert } Y[j] \\ S[i-1][j]+1 & \text{delete } X[i] \\ S[i-1][j-1] + \mathbb{1}(X[i] \neq Y[j]) & \text{sub } X[i] \to Y[j] \text{ not necessary if equal} \end{pmatrix}$$

Runtime: Again $O(mn)$.

Longest palindromic substring (Part III, Section 4.2)

Palindrome: "RACECAR" (odd length)

center

"TATTARRATTAT" (even length)

center x2

Input: String X

Output: $|Z|$, largest possible length of palindromic substring Z of X

$\boxed{\text{Example}}$

$$X = \text{"b}\underbrace{\text{anana}}_{\text{longest possible } Z}\text{"}$$

$$X = \text{"b}\underbrace{\text{anaana}}_{\text{longest possible } Z}\text{"}$$

Idea 1: DP

To determine $\underbrace{\text{whether } X[i:j] \text{ is palindrome}}_{:= S[i][j]}$

either:
- just check $(j \leq i+1)$
- $S[i][j] = S[i+1][j-1]$ AND $(X[i] == X[j])$

$O(n^2)$ time

Idea 2 (or 1?): center growing

1) guess center c or cc — $O(n)$
2) grow left & right pointers until can't — $\times O(n)$ $= O(n^2)$

Seems to match DP.

Manacher's algo: $O(n)$ (see notes)

Game theory (Part III, Section 5.1)

Consider two-player win-lose game.

Alice vs. Bob. E.g. Tic-tac-toe, chess, ...

Move 1 Alice

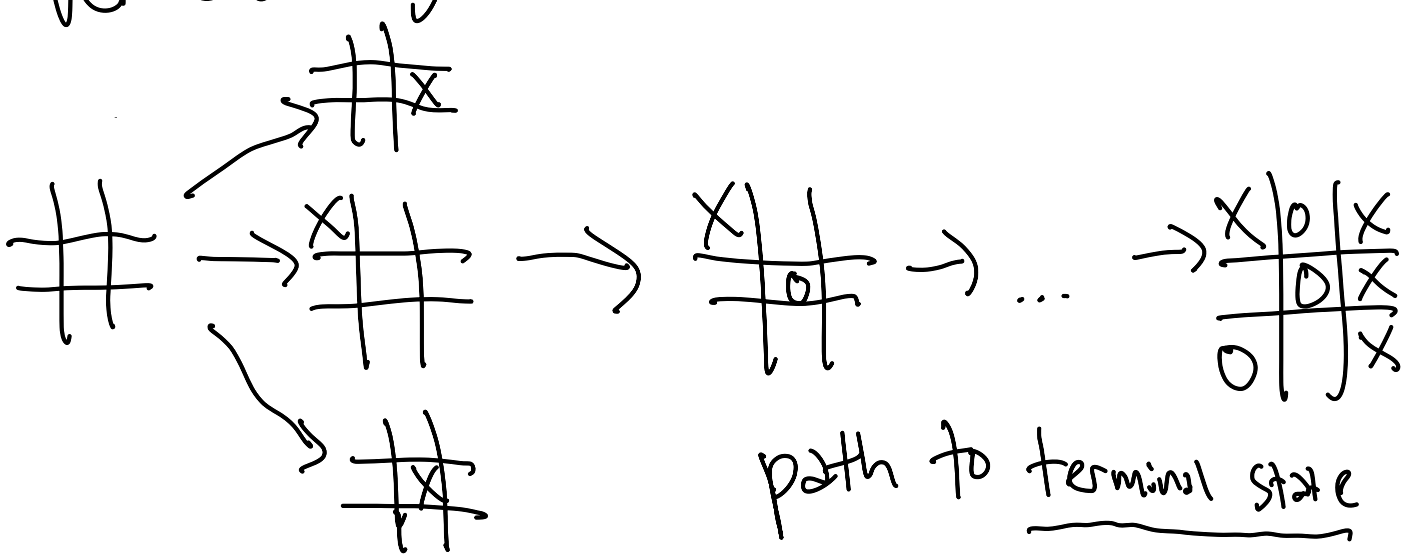
Move 2 Bob

⋮

Move k (game over, Alice wins)

Game graph (potentially huge, pruning in practice)

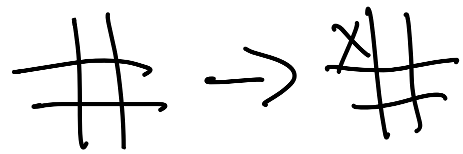
Vertices: game states



path to terminal state

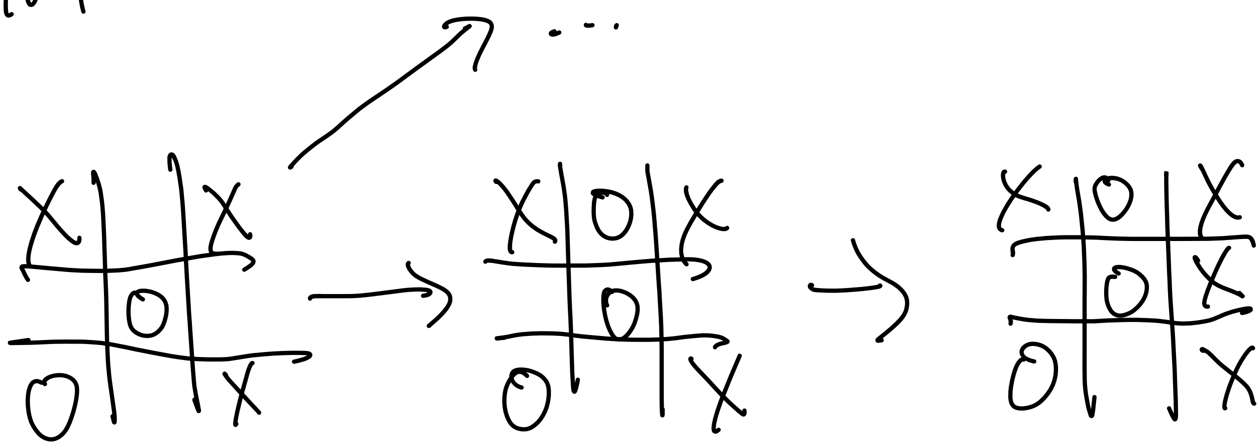
Terminal state: vertex where game is over.

1) Alice wins
2) Bob wins
3) Tie (we'll mostly ignore)

Game moves: directed edges 

Q: Can Alice always force a win?

Intuition:



"forced win"

Whatever Bob does, Alice can win.

DP solution: label all vertices v True "forced win"

False

Work backwards from leaves = terminal states
(next class)

(with ties, label them False.)

$$S(v) = \begin{cases} \underset{\text{edge }(v,u)}{\text{OR}}\left(S(u) \right) & \text{Alice plays} \\ \underset{\text{edge }(v,u)}{\text{AND}}\left(S(u) \right) & \text{Bob plays} \end{cases}$$

Alice wins if she can move to another winning state.

Bob wins if he can move to another losing state.

So Alice needs all possible moves to be True.

Extension

Zero-sum games

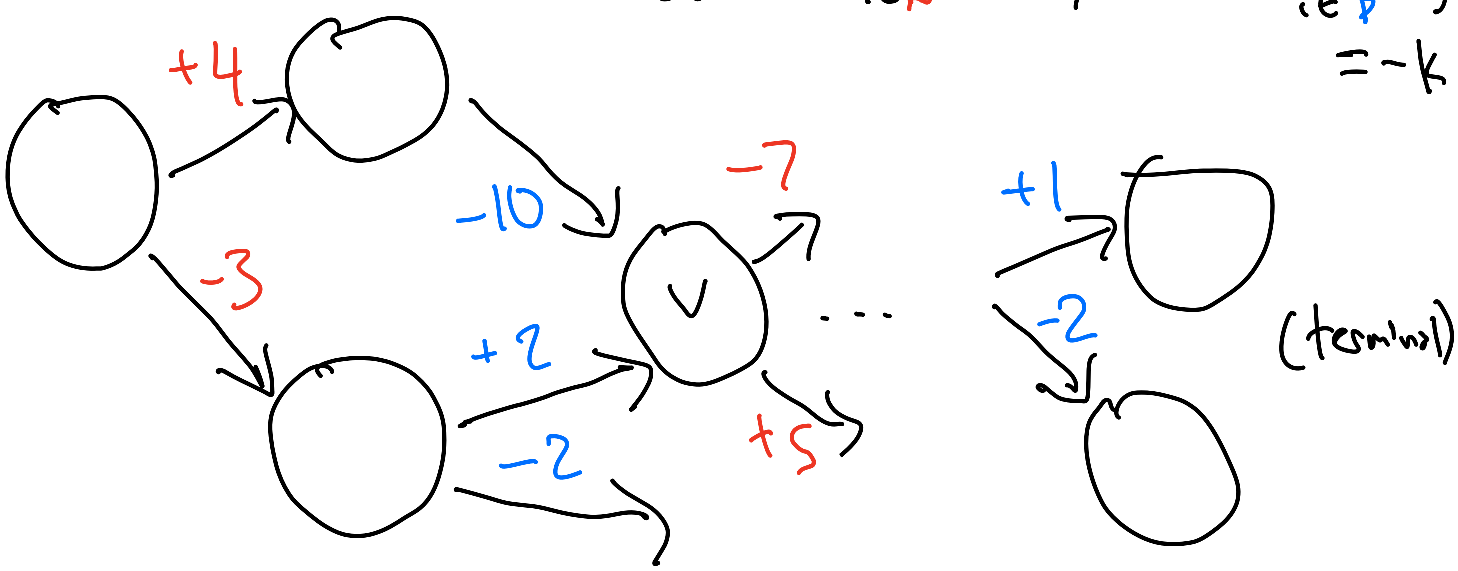
Alice and Bob have scores.

@ end of game, sum = 0.

e.g. win-lose (score at end is winner +1, loser −1)

dividing items ($v_1 + \dots + v_n = T$ split to A, B

score: $\sum_{i \in A} v_i = k$, $-T + \sum_{i \in B} v_i = -k$)



$$S[v] = \begin{cases} \max\limits_{\text{edge }(v,u)} S[u] + \text{score change}_{(v,u)} & \text{Alice} \\ \min\limits_{\text{edge }(v,u)} S[u] + \text{score change}_{(v,u)} & \text{Bob} \end{cases}$$

max guaranteed score if dropped in at state v.