Week 12 - Lecture Notes

Topics: Dynamic Programming

- memoization and subproblems
- fibonacci } Examples
- Shortest paths }
- guessing and DAG views

Computational Complexity

Dynamic Programming (DP)

- Big idea, hard, yet simple
- Powerful algorithmic design technique
- Large class of seemingly exponential problems have a polynomial solution ("only") via DP.
- Particularly for optimization problems (min/max)
  - **Example:** Shortest paths.

NPTEL

A dynamic programming is a controlled brute-force method.

It uses recursion and re-use.

ie.

DP ≈ "controlled- brute-force"

DP ≈ "recursion and re-use"

Fibonacci Numbers

Fibonacci numbers are of the form

$$f_1 = f_2 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

Goal: Compute F_n

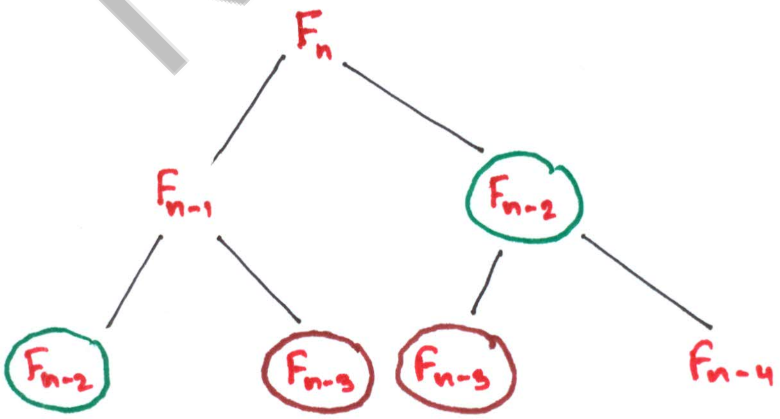
Naive Algorithm

follows recursive definition.

```
fib(n):
1. if $n \leq 2$ return $f = 1$
2. else return $f = fib(n-1) + fib(n-2)$
```

$$\Rightarrow T(n) = T(n-1) + T(n-2) + O(1)$$
$$\geq F_n \approx \phi^n$$
$$\geq 2T(n-2) + O(1)$$
$$\geq 2^{n/2}$$

Exponential - BAD!



NPTEL

Memoized DP Algorithm

```
memo = { }
fib(n):
    if n is in memo : return memo[n]
    else: if $n \le 2$ : $f = 1$
          else $f = fib(n-1) + fib(n-2)$
          memo[n] = f
          return f
```

- fib(k) only recurses first time called $\forall k$
- only n nonmemoized cells : $k = 1, 2, \ldots, n$
- memoized calls free ($\Theta(1)$ time)
- $\Theta(1)$ time per call (ignoring recursion)

Polynomial - GOOD!

- DP ≈ "recursion + memoization"
  - memoize (remember) and re-use solutions to subproblems that help solve problem
    - in Fibonacci, subproblems are $F_1, F_2, \ldots, F_n$

⟹ time = # subproblems · (time per subproblem)

- Fibonacci : # subproblems = n

  time per subproblem = $\Theta(1)$

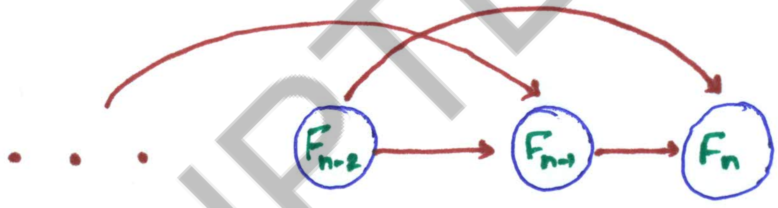
  ∴ time = $\Theta(n)$ (ignoring recursions)

NPTEL

Bottom-up DP Algorithm

```
fib = {}
for k in [1,2,...,n]:                                 ⎫
    if k ≤ 2: f = 1                   ⎫               ⎪
    else: f = fib[k-1] + fib[k-2]     ⎬ $\Theta(1)$   ⎬ $\Theta(n)$
    fib[k] = f                        ⎭               ⎪
return fib[n]                                         ⎭
```

- exactly the <u>same</u> computation as <u>memoized DP</u> (<u>recursion "unrolled"</u>)
- in general: <u>topological sort</u> of <u>subproblem dependency DAG.</u>



- practically <u>faster</u>: <u>no recursion</u>
- <u>analysis</u> more <u>obvious</u>
- can save space: <u>last 2 fibs ⇒ $\Theta(1)$</u>

NPTEL

Shortest Paths

- Recursive formulation

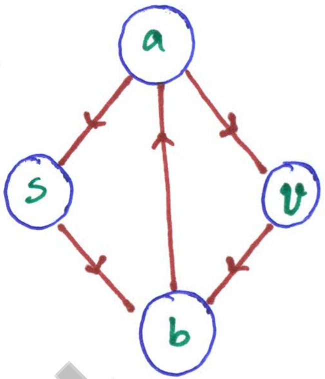
  $\delta(u,v) = \min\{w(u,v) + \delta(s,u) \mid (u,v) \in E\}$

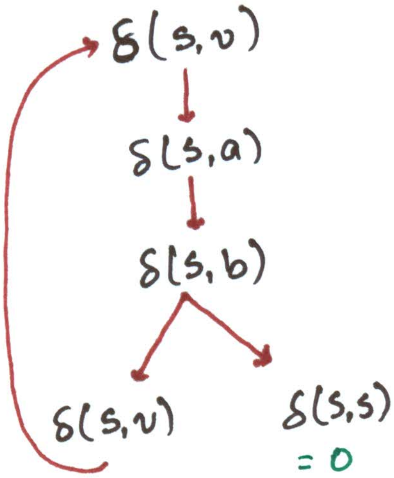
- Memoized DP algorithm : takes infinite time if cycles.

  (necessary to handle negative cycles)

- Works for directed acyclic graphs in $O(V+E)$

  ~(effectively DFS/topological sort + Bellman ford rolled into single recursion)





NPTEL

- Subproblem dependency should be acyclic.

  - more subproblems remove cyclic dependence

    $\delta_k(s,v)$ = shortest $s \to v$ path using $\leq k$ edges

  - recurrence :

    $\delta_k(s,v) = \min\{\delta_{k-1}(s,u) + w(u,v) \mid (u,v) \in E\}$

    $\left.\begin{array}{l} \delta_0(s,v) = \infty \text{ for } s \neq v \\ \delta_k(s,s) = 0 \text{ for any } k \end{array}\right\}$ base case

  - goal : $\delta(s,v) = \delta_{|V|-1}(s,v)$ } if no negative cycle exists

  - memoize

  - time : $\underbrace{\#\text{ subproblems}}_{|V||V|} \cdot \underbrace{(\text{time per subproblems})}_{O(v)} = O(V^3)$

  - actually $\Theta(\text{indegree}(v))$ for $\delta_k(s,v)$

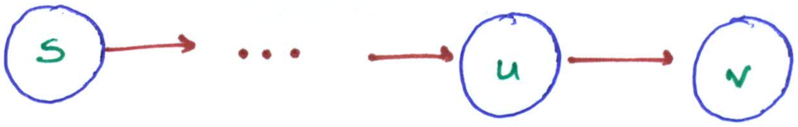
  $\Longrightarrow$ time $\Theta(V \sum_{v \in V} \text{indegree}(v)) = \Theta(VE)$

BELLMAN FORD!

Guessing

How to design recurrence

- want shortest s → v path



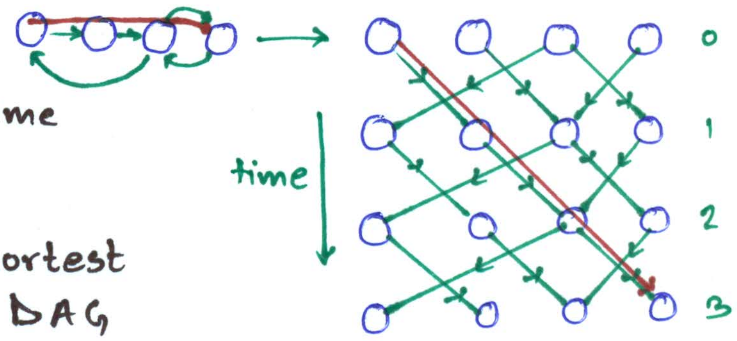
- what is the last edge in path? don't know
- guess it is (u,v)
- path is $\underbrace{\text{shortest } s \to u \text{ path}}_{\text{by optimal substructure}} + \text{edge } (u,v)$
- cost is $\underbrace{\delta_{k-1}(s,u)}_{\text{another subproblem}} + w(u,v)$
- to find best guess, try all (|V| choices) and use best.
- *Key: small (polynomial) # possible guesses per subproblem
  - typically this dominates time/subproblem.

* DP ≈ recursion + memoization + guessing

NPTEL

DAG view

- like replicating graph to represent time
- converting shortest paths in graph to shortest paths in DAG



* DP ≈ shortest paths in some DAG

Summary

- DP ≈ careful brute force
- ≈ guessing + recursion + memoization
- ≈ dividing into reasonable # subproblems whose solution relate - acyclicly - usually via guessing parts of solution

- time = # subproblems × $\underbrace{\text{(time per subproblem)}}_{\text{treating recursive calls as } O(1)}$

  (usually mainly guessing)

  - essentially an amortization
  - count each subproblem only once ; after first time, costs O(1) via memoization

- DP ≈ shortest paths in some DAG.

NPTEL

5 easy steps to Dynamic Programming

define subproblems	count # subproblems
guess (part of solution)	count # choices
relate subproblem solutions	compute time per subproblem
recurse + memoize problems OR build DP table bottom-up check subproblems acyclic/topological order.	time = (time per subproble) × # subproblems
solve original problem: = a subproblem OR by counting subproblem solutions.	⟹ extra time

Examples	Fibonacci	Shortest paths
subproblems	F_k for $1 \le k \le n$	$\delta_k(s,v)$ for $v \in V$, $0 \le k \le \lvert V \rvert$ = min $s \to v$ path using $\le k$ edges
#subproblems	n	V^2
guess	nothing	edge into v (if any)
# choices	1	indegree$(v)+1$
recurrence	$F_k = F_{k-1} + F_{k+2}$	$\delta_k(s,v) = \min\{\delta_{k-1}(s,u) + w(u,v) \mid (u,v) \in E\}$
time per subproblem	$\Theta(1)$	$\Theta(1 + \text{indegree}(v))$
topological order	for $k=1,\dots,n$	for $k = 0,1, \dots \lvert V \rvert - 1$ for $v \in V$
total time	$\Theta(n)$	$\Theta(VE)$ $+ \Theta(V^2)$ unless efficient about indegree
original problem	F_n	$\delta_{\lvert V \rvert -1}(s,v)$ for $v \in V$
extra time	$\Theta(1)$	$\Theta(V)$

NPTEL

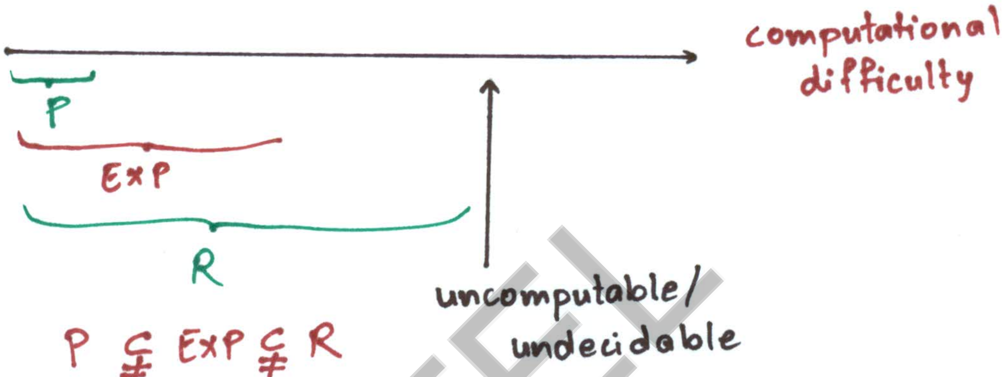
Computational Complexity

Definitions:

P = { problems solvable in (n^c) time } (polynomial)

EXP = { problems solvable in (2^{n^c}) time } (exponential)

R = { problems solvable in finite time } "recursive"



$P \subsetneq EXP \subsetneq R$

Examples:

negative-weight cycles detection $\in P$

n×n Chess $\in EXP$ but $\notin P$

↳ who wins from given board configuration?

Tetris $\in EXP$ but don't know whether $\in P$

↳ survive given pieces from given board.

Halting Problem

Given a computer program, does it ever halt (stop)?

- uncomputable ($\notin R$) : no algorithm solves it (correctly in finite time on all inputs)
- decision problem : answer is YES or NO

NPTEL

Most Decision Problems are Uncomputable

- program ≈ binary string ≈ nonnegative integer $\in \mathbb{N}$
- decision problem = a function from binary strings (≈ non-neg. intergers) to {YES (1), NO (0)}
- ≈ infinite sequence of bits ≈ real number $\in \mathbb{R}$
  $|\mathbb{N}| \ll |\mathbb{R}|$: no assignment of unique nonnegative integers to real numbers ($\mathbb{R}$ uncountable)
- ⇒ not nearly enough programs for all problems
- each program solves only one problem
- ⇒ almost all problems cannot be solved

NP

NP = {Decision problems solvable in polynomial time via a lucky algorithm} "The lucky algorithm can make lucky gueses, always "right" without trying all options"

- nondeterministic model : algorithm makes gueses and then says YES or NO
- guesses guaranteed to lead to YES outcome if possible

Example:

Tetris $\in$ NP

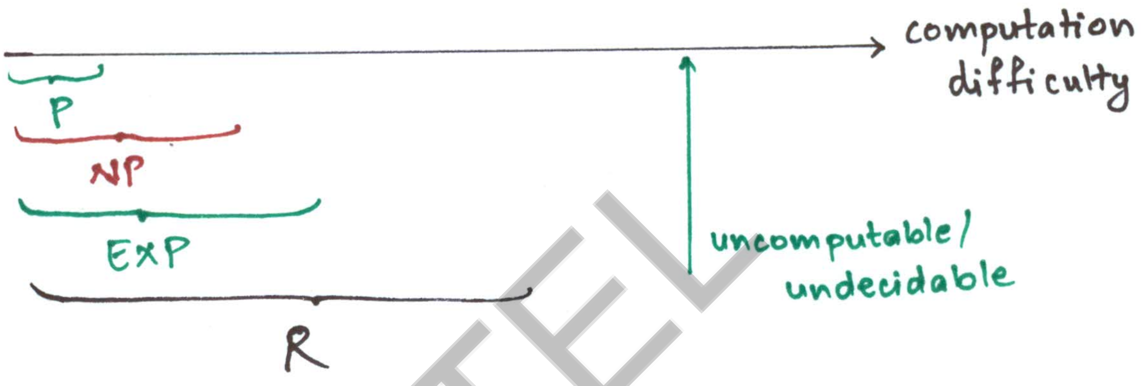
- nondeterministic algorithm : guess each move, did I survive ?
- proof of YES : list what moves to make (rules of Tetris are easy)

NPTEL

NP

NP = {decision problems with solutions that can be "checked" in polynomial time}

⇒ when answer is YES,
it can be proved, and
polynomial-time algorithm can check proof.



NPTEL

P ≠ NP

It is a big conjecture (worth $1,000,000)

- ≈ cannot engineer luck
- ≈ generating (proofs of) solutions can be harder than checking them

Hardness and completeness

Claim:

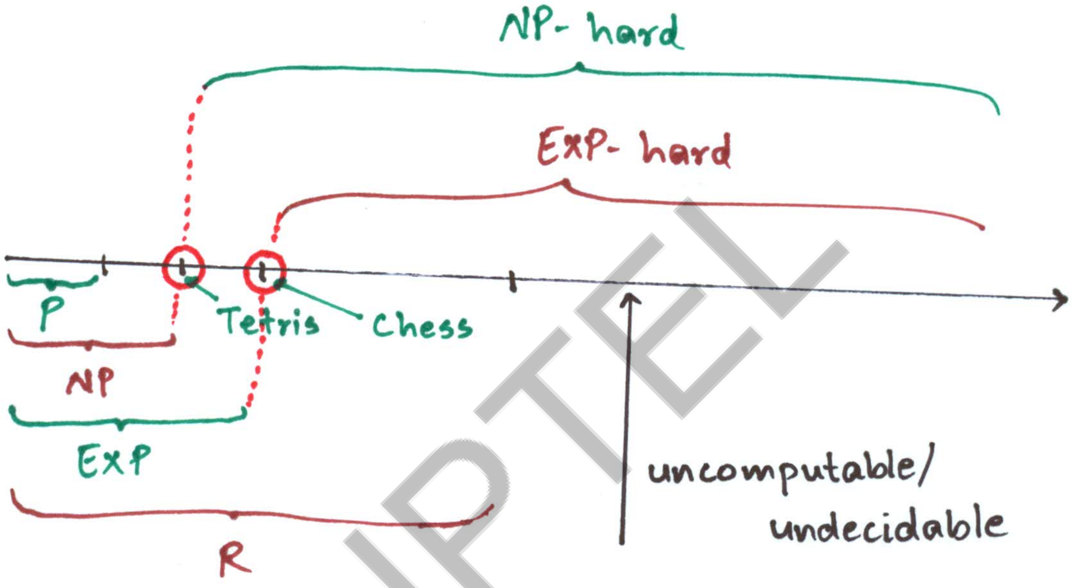
If $P \neq NP$, then Tetris $\in NP - P$

Proof:

- Tetris is NP-hard = "as hard as" every problem $\in NP$

Infact

- Tetris is NP-complete = $NP \cap (NP\text{-hard})$



NPTEL

- Chess is EXP-complete = $EXP \cap EXP\text{-hard}$.

EXP-hard is as hard as every problem in EXP.

If $NP \neq EXP$, then Chess $\notin EXP \setminus NP$.

Whether $NP \neq EXP$ is also an open problem but "less famous / "important".

Reductions

Convert the problem into a problem that is already known how to solve (instead of solving from scratch)

- most common algorithm design technique
- unweighted-shortest path → weighted (set weights = 1)
- min product path → shortest path (take logs)
- longest path → shortes path (negative weights
- shortest order tour → shortest path (k copies of the graph)
- cheapest leaky-tank path → shortest path (graph reduction)

NPTEL

All of the above are One-call reductions:

A problem → B problem → B solution → A solution

Multicall reductions:

- solve A using free calls to B,

"in this sense, every algorithm reduces problem → model of computation."

NP- Complete Problems

NP- Complete problems are all interreducible using polynomial time reductions (same difficulty)

We can use reductions to prove NP-hardness → Tetris.

Examples of NP- Complete Problems

- Knapsack
- 3- partition : given n integers, divide them into triples of equal sum?
- Travelling Salesman Problem:
  - → shortest path that visits all vertices of a given graph
  - → is minimum weight $\leq x$? (decision version)
- longest common subsequence of k strings
- Minesweeper, Soduku and most puzzles
- SAT : given a Boolean formula (and, or, not), is it ever true ?
- Shortest paths amidst obstacles in 3D
- 3- coloring a given graph
- find largest clique in a given graph.

NPTEL