# CS3233
# Competitive Programming

Dr. Steven Halim

## Week 06 – Problem Solving Paradigms
## (Dynamic Programming 2)

# Outline

- Mini Contest #5 + Break + Discussion + Admins
- A simple problem to refresh our memory about DP paradigms
- DP and its relationship with DAG (Chapter 4)
  - DP on Explicit DAG/Implicit DAG
    - These are CS2020/CS2010 materials
    - Those from CS1102 must catch up/consult Steven separately
- DP on Math Problems (Chapter 5)
- DP on String Problems (Chapter 6)
- DP + bitmask (Chapter 8)
- Compilation of Common DP States

DP Problems on Chapter 4-5-6 of CP2 Book

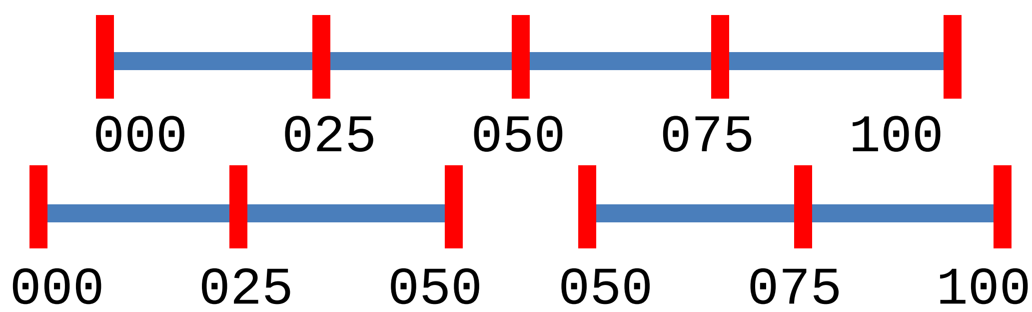
# NON CLASSICAL DP PROBLEMS

# Non Classical DP Problems (1)

- **My** definition of *Non Classical DP* problems:
  - Not the pure form/variant of LIS, Max Sum, Coin Change, 0-1 Knapsack/Subset Sum, TSP where the DP **states** and **transitions** can be "memorized".
  - Requires **original formulation** of DP states and transitions
    - Although some formulation are still "similar" to the classical ones
  - Throughout this lecture, we will talk mostly in *DP terms*
    - **State** (to be precise: "*distinct* state")
    - **Space Complexity** (i.e. the number of distinct states)
    - **Transition** (which entail overlapping sub problems)
    - **Time Complexity** (i.e. num of distinct states * time to fill one state)

# Non Classical DP Problems (2)

- To refresh our memory ☺
- Example: Cutting Sticks (UVa 10003) – in CLRS 3rd ed!
  - State: dp[l][r], Q: Why these two parameters?
  - Space Complexity: Max 50x50 = 2500 distinct states
  - Transition: Try all possible cutting points **m** between **l** and **r**,
    - i.e. cut (l, r) into (l, m) and (m, r)
  - Time Complexity: There can be up to 50 possible cutting points, thus max 2500 x 50 = 125000 operations, do able ☺



# DP on DAG (1)

## Overview

- Dynamic Programming (DP) has a close relationship with (sometimes implicit) Directed Acyclic Graph (DAG)
  - The **states** are the **vertices** of the DAG
  - Space complexity: Number of vertices of the DAG
  - The **transitions** are the **edges** of the DAG
    - Logical, since a recurrence is always **acyclic**
  - Time complexity: Number of edges of the DAG
  - Top-down DP: Process each vertex just once via **memoization**
  - Bottom-up DP: Process the vertices in **topological order**
    - Sometimes, the topological order can be written by just using simple (nested) loops

# DP on DAG (2)

## On Explicit DAG (1)

- There is a DAG in this problem (can you spot it?)
  - UVa 10285 (Longest Run on a Snowboard)
- Given a R*C of height matrix h,
  find what is the longest run possible?
  - Longest run of length k:
    - $h(i_1, j_1) > h(i_2, j_2) > \ldots > h(i_k, j_k)$
  - **Longest path in DAG!**
- We will learn a creative DP table filling technique from this short example (bottom up)

# DP on DAG (2)

## On Explicit DAG (2)

- Complete search recurrence:
  - Let h(i, j) = height of location (i, j)
  - Let f(i, j) = longest run length started at (i, j)
  - f(i, j) = 1 + max(f values of reachable neighbors: h(i, j) > h(neighbors' i, j))
    - Base case: If h(i, j) is the lowest among its neighbor, then f(i, j) = 1
- Do it from all possible R*C locations!
  - Do you observe many overlapping sub problems?
- This problem is actually solvable using complete searchdue to "small" R*C + constraints!
  - But with DP, the solution is more efficient
    - And it can be used to solve larger R*C

# DP on DAG (2)

## On Explicit DAG (3)

- Fill-in-the-Table (Bottom-Up) Trick:
  - Sort (i, j) according to h(i, j) in **increasing order**
    - i.e. Fill the bottom cells first!
    - Fill in f(i, j) in order based on the value of f(i-1, j), f(i+1, j), f(i, j-1), f(i, j+1)
      - This is the topological sort!
  - This may actually be harder to code
    - For this problem, writing the DP solution in top-down fashion may be better/easier

# DP on DAG (3)

## Converting a General Graph → DAG (1)

- Not every graph problem has a ready algorithm for it
  - Some have to be solved with Complete Search…
  - Some are solvable with DP, as long as we can find a way to make the **DP transition acyclic**, usually by adding one (or more) parameters to each vertex :O
  - Remember that Dijkstra's is a greedy algorithm, and every greedy solution (without proof of correctness) has a chance for getting WA…
- Example: Fishmonger (SPOJ 101)
  - State: dp[pos][time_left], Q: Why these two parameters?
  - Space Complexity: Max 50 * 1000 = 50000
  - Transition: Try all remaining N cities, notice that time_left always decreases as we move from one city to another (**acyclic!!**)
  - Time Complexity: Max 50000 * 50 = 2.5 M, doable ☺

# DP on Math Problems (Chapter 5)

- Some well-known mathematic problems involves DP
  - Some combinatorics problem have recursive formulas which entail overlapping subproblems
    - e.g. those involving Fibonacci number, **f(n) = f(n - 1) + f(n -2)**
  - Some probability problems require us to search the entire search space to get the required answer
    - If some of the sub problems are overlapping, use DP, otherwise, use complete search
  - Mathematics problems involving **static** range sum/min/max!
    - Use dynamic tree DS for dynamic queries

# DP on String Problems (Chapter 6)

- Some string problems involves DP
  - Usually, we do not work with the string itself
    - Too costly to pass (sub)strings around as function parameters
  - But we work with the integer indices to represent suffix/prefix/substring
- Example: UVa 11258 – String Partition
  - There are many ways to split a string of digits into a list of non-zero-leading (0 itself is allowed) 32-bit *signed* integers
    - That is, max integer is $2^{31}$-1 = 2147483647
  - What is the maximum sum of the resultant integers if the string is split appropriately?

Last part of this lecture, from Chapter 8 of CP2 Book

# DP + BITMASK AND TIPS/TRICKS

# Emerging Technique: DP + bitmask

- We have seen this form earlier in DP-TSP
- Bitmask technique can be used to represent *lightweight set of Boolean* (up to $2^{64}$ if using unsigned long long)
- Important if one of the DP parameter is a "set"
- Can be used to solve ***matching in small general graph***
- Example: Forming Quiz Teams (UVa 10911)
  - State: dp[bitmask]
  - Space Complexity: $2^M$ ~ 65K distinct sub problems; max N = 8, M = 2N, thus max M = 16
  - Transition: Clever version: O(N), try all and break as soon as we find the first perfect matching, Not so clever: $O(N^2)$
  - Time Complexity: $O(N.2^M)$ or about 524K, doable

# Common DP States (1)

- Position:
  - Original problem: $[x_1, x_2, \ldots, x_n]$
    - Can be sequence (integer/double array), can be string (char array)
  - Sub problems, break the original problem into
    - Sub problem and Prefix: $[x_1, x_2, \ldots, x_{n-1}] + x_n$
    - Suffix and sub problem: $x_1 + [x_2, x_3, \ldots, x_n]$
    - Two sub problems: $[x_1, x_2, \ldots, x_i] + [x_{i+1}, x_{i+2}, \ldots, x_n]$
  - Example: LIS, 1D Max Sum, Matrix Chain Multiplication (MCM), etc

# Common DP States (2)

- Position**s**:
  - This is similar to the previous slide
  - Original problem: $[x_1, x_2, \ldots, x_n]$ and $[y_1, y_2, \ldots, y_n]$
    - Can be two sequences/strings
  - Sub problems, break the original problem into
    - Sub problem and prefix: $[x_1, x_2, \ldots, x_{n-1}] + x_n$ and $[y_1, y_2, \ldots, y_{n-1}] + y_n$
    - Suffix and sub problem: $x_1 + [x_2, x_3, \ldots, x_n]$ and $y_1 + [y_2, y_3, \ldots, y_n]$
    - Two sub problems: $[x_1, x_2, \ldots, x_i] + [x_{i+1}, x_{i+2}, \ldots, x_n]$ and
      $[y_1, y_2, \ldots, y_i] + [y_{i+1}, y_{i+2}, \ldots, y_n]$
  - Example: String Alignment/Edit Distance, LCS, etc
  - PS: Can also be applied on 2D matrix, like 2D Max Sum, etc

# Tips: When to Choose DP

- Default Rule:
  - If the given problem is an **optimization** or **counting** problem
    - Problem exhibits optimal sub structures
    - Problem has overlapping sub problems
- In ICPC/IOI:
  - If actual solutions are not needed (only final values asked)
    - If we must compute the solutions too, a more complicated DP which stores *predecessor information* and *some backtracking* are necessary
  - The number of distinct sub problems is small enough (< 1M)
    and you are not sure whether greedy algorithm works (why gamble?)
  - Obvious overlapping sub problems detected :O

# Dynamic Programming Issues (1)

- Potential issues with DP problems:
  - They may be disguised as (or looks like) non DP
    - It looks like greedy can work but some cases fails…
      - e.g. problem looks like a shortest path with some constraints on graph, but the constraints fail *greedy* SSSP algorithm!
  - They may have subproblems but not overlapping
    - DP does not work if overlapping subproblems not exist
      - Anyway, this is still a good news as perhaps Divide and Conquer technique can be applied

# Dynamic Programming Issues (2)

- Optimal substructures may not be obvious
  1. Find correct “states” that describe problem
     - Perhaps extra parameters must be introduced?
  2. Reduce a problem to (smaller) sub problems (with the same states) until we reach base cases
- There can be more than one possible formulation
  - Pick the one that works!

# DP Problems in ICPC (1)

- The number of problems in ICPC that must be solved using DP are growing!
  - At least one, likely two, maybe three per contest…
- These new problems are **not** the classical DP!
  - They require deep thinking…
  - Or those that look solvable using other (simpler) algorithms but actually must be solved using DP
  - Do not think that you have "mastered" DP by only memorizing the classical DP solutions!

# DP Problems in ICPC (2)

- In 1990ies, mastering DP can make you "king" of programming contests…
  - Today, it is a must-have knowledge…
  - So, get familiar with DP techniques!
- By mastering Graph + DP, your ICPC rank is probably:
  - from top ~[25-30] (solving 1-2 problems out of 10)
    - Only easy problems
  - to top ~[10-20] (solving 3-5 problems out of 10)
    - Easy problems + some graph + greedy/DP problems

For Week 07 homework ☺

(You can do this over recess week too)

# BE A PROBLEM SETTER

# Be a Problem Setter

- Problem Solver:
  A. Read the problem
  B. Think of a good algorithm
  C. Write 'solution'
  D. Create tricky I/O
  E. If WA, go to A/B/C/D
  F. If TLE/MLE, go to A/B/C/D
  G. If AC, stop ☺
- Problem Setter:
  A. Write a good problem
  B. Write good solutions
     - The correct/best one
     - The incorrect/slower ones
  C. Set a good secret I/O
  D. Set problem settings
- A problem setter must think from a different angle!
  – By setting good problems, you will simultaneously be a better problem solver!!

# Problem Setter Tasks (1)

- Write a <u>good</u> problem
  - Options:
    - Pick an algorithm, then find problem/story, or
    - Find a problem/story, then identify a good algorithm for it (harder)
  - Problem description must not be <u>ambiguous</u>
    - Specify input constraints
    - Good English!
    - Easy one: longer,<br>Hard one: shorter!
- Write <u>good</u> solution<u>**s**</u>
  - Must be able to solve your own problem!
    - To set hard problem, one must increase his own programming skill!
  - Use the <u>best possible</u> algorithm with lowest time complexity
    - Use the inferior ones 'that barely works' to set the WA/TLE/MLE settings...

# Problem Setter Tasks (2)

- Set a good secret I/O
  - Tricky test cases to check AC vs WA
    - Usually ‘boundary case’
  - Large test cases to check AC vs TLE/MLE
    - Perhaps use input generator to generate large test case, then pass this large input to our correct solution
- Set problem settings
  - Time Limit:
    - Usually 2 or 3 times the timings of your own best solutions
      - Java slower than C++!
  - Memory Limit:
    - Check OJ setting^
  - Problem Name:
    - Avoid revealing the algorithm in the problem name

# FYI: Be A Contest Organizer

- Contest Organizer Tasks:
  - Set problems of *various* topic
    - Better set by >1 problem setter
  - Must balance the difficulty of the problem set
    - Try to make it fun
    - Each team solves some problems
    - Each problem is solved by some teams
    - No team solve all problems
      - Every teams must work until the end of contest

# Special Homework for Week07

- Create **one** problem of your choice
  - Can be of **any** problem type
    - Regardless we have studied that in CS3233 or not
    - For those who are new with Competitive Programming, just create one from these: Ad Hoc/Libraries/BF/D&C/Greedy/DP/simple Graph
    - You can do this in advance (use your mid sem break?)
- Deliverables:
  - 1 html: problem description + sample I/O
  - 1 source code: cpp/java
  - 1 test data file (of multiple instances type)
  - 1 short txt write up of the expected solution
  - Zip and upload your problem into special folder in IVLE
    - CS3233 / Homework / Be A Problem Setter

# More References

- **Competitive Programming 2**
  - Section 3.5, 4.7.1, 5.4, 5.6, 6.5, and 8.4
- **Introduction to Algorithms**, p323-369, Ch 15
- **Algorithm Design**, p251-336, Ch 6
- **Programming Challenges**, p245-267, Ch 11
- http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=dynProg
- Best is practice & more practice!