Dynamic Programming IV [MIT Open Courseware 6.006]

- 5 easy steps to DP:
  1. define subproblems
     # subproblems ?
  2. Guess / Try part of solution
     # guesses ?
  3. Recurrence
  4. Recurse + Memoize
  5. Solve original problem

- **Question:** What types of guessing have we seen?
  - Prefix
  - Suffix
  - Substring

- Today we will see an additional types
- In step 2 we are guessing which subproblem to use in order to solve the original problem

- But there is a higher-level:

  In step 1 we can add more subproblems to guess/remember more

  This happend in the knapsack problem:

  - The obvious solution was to use affixes;
  - But using affixes alone was insufficient to solve the problem
  - We also need the ability to remember how much capacity remained.

- Lets look at some examples

Piano / Guitar fingering

- Given sequence of n notes, find fingering for each note
- By fingering we mean: which finger should be used to play each note since some transitions are easier than others.

- For humans we have 1, 2, 3, ..., 10
- Generalizing, fingers : 1, ..., F
- To keep it simple assume:
  - Only a single note on the piano
  - Only the right-hand is used
- **Objective:**

  Assign one of each finger to each node

- But we also need to express the difficulty of beings on a certain note that is being played with a certain finger and we want to transition another note using another finger.
- **Question:**

  How can we express this difficulty?

- We can define a function:

  $d(p, f, q, g)$:

  - p – first note
  - f – finger that is playing note p
  - q – note that we want to transition
  - g – finger that will play note q

- There is a <u>huge literature</u> for this on piano!, e.g.:
  - If $p \ll q$ then you need to stretch your fingers a lot, which is hard!
  - If you are playing <u>legato</u> the player transitions from one note to another note with no intervening silence:
    - You cannot use the same finger for the same note
    - This implies that if $f == g$ then $p = q$
  - Weak finger rule: avoid the two rightmost fingers (fingers 4 & 5) of ~~right~~ right-hand
  - Many more other examples, including for guitar
- These transitions can be encoded into a table that represents function <u>d</u>

- Lets try to solve this using DP:
  - We have a sequence of notes and we want to find the sequence of fingers

Question:

- If we have a sequence of something what can we try?

  ↳ Always the same answer: { prefixes, suffixes, substrings

Question:

which one do you think it will be?

- Intuitively: We want to process the sequence of notes from left to right ⟹ suffixes

Notes: This initial formulation is wrong! It is used to exemplify a problem

① Subproblem = how to play notes[i:]

② guess = which finger to use for note i

③ recurrence: (we want to minimize difficulty)

$$DP(i) = \min\Big(\underbrace{DP[i+1]}_{\text{play remaining notes}} + d(i, f, i+1, ?)\ \underbrace{\text{for } f \text{ in } 1, \dots, F}_{\text{for all the fingers}}\Big)$$

We are playing note i with finger f then we have to go to note $i+1$ but then the problem is we have no idea what finger we are going to guess for not $i+1$

∴ We cannot solve the problem this way!!

⟶ But it was the first thing we should try

– We need to add more subproblems...

Question: Any guesses to what we can do for subproblems?

① Subproblems = how to play notes[i:] when using finger f for notes[i]

#notes = n
#fingers = F } ⇒ #subproblems = n. F

② Try/Guess = what is the finger g for notes[i+1]

#guesses = "for each note we need to try all fingers" = F

③ Recurrence

notes[i] notes[i+1]

$$DP(i,f) = \min\left(DP(i+1, g) + d(i, f, i+1, g) \text{ for all } g \text{ in } 1, \dots, F \right)$$

i:= which note am I in
f:= what is the finger for note i

④ (Always the same thing)

⑤ Original problem:

– We cannot just start in any finger;

– We need to select the finger that minimizes difficulty

– I.e.:

$$\min\left(DP(0, f) \text{ for } f \text{ in } 1, \dots, F \right)$$

Question: What is the running time?

- #subproblems = nF
- # guesses = F (time / subproblem)
- Determining initial finger = F
- Total time: $n \cdot F \cdot F \cdot F = nF^3$
  - Given that $\underline{F}$ is usually pretty small this is approximately linear time, i.e. $\Theta(n)$
- Lets look at another subproblem

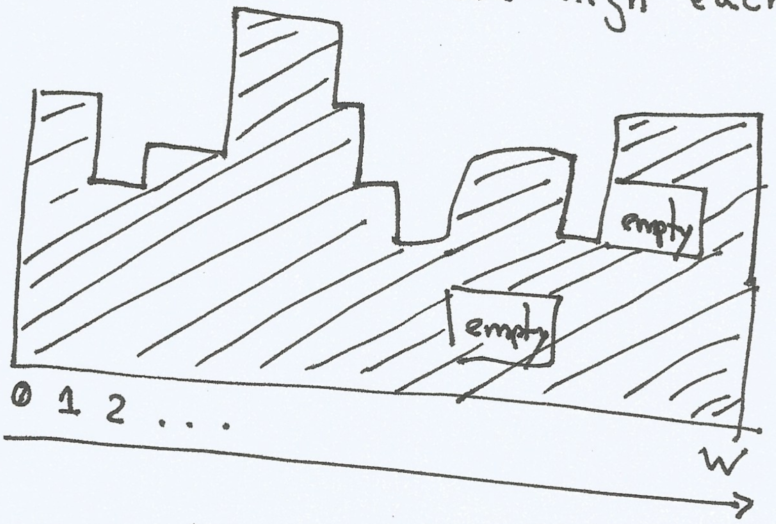
__Tetris training__ – (restrained / constrained version)

__Assume:__
- Given sequence of $\underline{n}$ pieces
- For each of them we must drop the piece from the top
- No rotations are allowed whilst the piece is falling
- Full rows normally clear, but in our version they will not clear → hardcore tetris ;)
- The width of the board $\underline{w}$ is small
- The board is initially empty

- We need all these assumptions to be able to use dynamic programming

① Subproblems:

- Obvious thing to try is suffixes;
- How to play suffix pieces[i:] (initial subproblem formulation)
- Just like the fingering problem this is not enough information
- We need to know what the board looks like
- All we care about is how high each column is:



This is called the skyline

- The new subproblem formulation:

  How to play suffix pieces[i:] given board skyline

- **Question:** How many possible configurations do we have for the skyline?
  - height of the board := h (from 0 to h)
  - Therefore we have: (h+1) combinations/spans for 1st column

    (h+1) " " " 2nd "

    ⋮

    (h+1) " " " " w^{th} column

$\therefore$ We have $\underbrace{(h+1)\times(h+1)\times\dots\times(h+1)}_{w\text{ times}} = (h+1)^w$ combinations

- For each one of the suffixes we need to take into account the skyline
- This implies that for $\underline{n}$ we need to calculate $(h+1)^w$
- #subproblems $= n.\ (h+1)^w$ $\underbrace{\left(\text{exponential in } \underline{w}\right)}$ This is why we need $\underline{w}$ to be small

② **Guess / Try**

- How to play piece $\underline{i}$
  - Normal tetris we have 4 rotations possible
  - For each colum $\underline{w}$ we need to determine the best rotation
  - # guesses $= 4.\ w$

③ **Recurrence:**

- Goal: place each piece in sequence to survive knowing the skyline, i.e.: the height of each colum.
- Let h_k - represent the height of colum $\underline{k}$
- Can we survive or not?
  - This can be represented by a binary value (0 or 1)
  - We can try all possiblities and perform a logical $\underline{OR}$ operation to see if we survive
- I.e. we can have the following recurrence

$$DP(i, h_1, \ldots, h_w) = \text{or}\ \{ DP(i+1, \overbrace{h_1', \ldots, h_w'}^{\text{how the skyline changed after placing piece } i}) ,\ \text{for each possible placement of piece } \underline{i} \}$$

④ Always the same thing

⑤ Original problem: $DP(\emptyset, \underbrace{\emptyset, \emptyset, \ldots, \emptyset}_{})$

first piece   empty board ⇒ skyline = ∅

Question: What is the total running time?

$\#$ subproblems $= n(h+1)^w$

$\#$ guesses / trys $= 4 \cdot w$

Total time $= n(h+1)^w \cdot 4w = \Theta(nw(h+1)^w)$

– Lets look at another problem

Super Mario Bros

- Given the entire level with n bits of information
- Small $w \times h$ screen
- We can solve Super Mario Bros by dynamic programming
- Various performance metrics can be used:
  - Run through a level and maximize yourscore
  - " " " " " minimize your time
  - Pick your favourite measure
- We ~~can~~ need to write down what do we need to know about the game state, lets call this the configuration
  - Current position of:
    - Monsters
    - Objects
    - Powerufs

Question:

- How much information do we need to store the ~~\$~~ configuration?

- Somewhere along the line of $c^{w.h}$, where c is a constant
  - Not very scientific, just a rough bound
  - Idea: for every pixel on the screen
    - Is the pixel a brick?
      - Is it a hard brick?
      - Is it a destroyed brick?
    - Is a monster there right now?
    - Is Mario there right now?

- We have a constant amount of choices per pixel
  - c choices for pixel 1;
  - c " " " 2;
  - ⋮ ⋮ ⋮ ⋮
  - c choices for pixel $w \times h$;

I.e. $\underbrace{c \times c \times \dots \times c}_{w \times h} = c^{w.h}$

- Succinctly, the configuration will have:
  - everything on screen: $c^{w.h}$
  - Mario velocity: c (assume constant amount of velocity choices)
  - How far have we gone to the right: w
  - score: S
  - time: T

  These two can be quite large integers so this hints that we will have a pseudo-polynomial

- The total number of configuration: $c^{w.h} \cdot w \cdot c \cdot S \cdot T = \Theta(w\,c^{w.h} \cdot S \cdot T)$

since this is a constant it can be discarded

- For every configurations there is a sequence of possible actions ~~to~~ that can be performed
- Our recurrence would need to maximize/minimize these actions for all possible configurations
- Obviously we are not going to write the recursion for this !! :)
- But because we know the total number of configurations and we have a constant amount of actions to try/guess per configuration, the total running time is: $\boxed{\Theta(w \cdot c^{w.h} \cdot S \cdot T)}$

pseudo-polynomial time