UNIT 7: DYNAMIC PROGRAMMING

L19. MEMOIZATION, SUBPROBLEMS, GUESSING, BOTTOM-UP; FIBONACCI, SHORTEST PATHS

DYNAMIC PROGRAMMING

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) (e.g., shortest paths)

*** DP ≈ "controlled brute force"**

*** DP ≈ recursion + re-use**

FIBONACCI NUMBERS (MEMOIZATION)

Naive Algorithm

```
fib(n):
  if n ≤ 2: return f = 1
  else: return f = fib(n - 1) + fib(n - 2)
```

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

Memoized DP Algorithm

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

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

*** DP ≈ recursion + memoization**

Bottom-up DP Algorithm

```
fib = {}
for k in [1, 2, . . . , n]:
  if k ≤ 2: f = 1
  else: f = fib[k − 1] + fib[k − 2]
  fib[k] = f
return fib[n]
```

- exactly the same computation as memoized DP (recursion "unrolled")
- in general: topological sort of subproblem dependency DAG
- practically faster: no recursion
- can save space: just remember last 2 fibs $\Longrightarrow \Theta(1)$

SHORTEST PATHS (GUESSING)

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

Guessing

- want shortest $s \to v$ path
- what is the last edge in path? dunno
- guess it is (u,v)
- path is $\underbrace{\text{shortest } s}_{\text{by optimal substructure}}$ $\to$ 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

<u>* DP ≈ recursion + memoization + guessing</u>

DAG view

Memoized DP algorithm: takes infinite time if cycles!

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

* DP ≈ shortest paths in some DAG

THINKING

本讲开始介绍Dynamic Programming（动态规划）。某种程度上就是递归思想的发展。有两个主要的技巧：

1. Memoization（记忆化）：对于子结果需要多次提取时
2. Guessing：实际上就是对于子问题的贪婪搜索（遍历）

R19. DYNAMIC PROGRAMMING: CRAZY EIGHTS, SHORTEST PATH

OPTIMAL SUB-STRUCTURE

DP takes the advantage of the *optimal sub-structure* of a problem. A problem has an optimal substructure if the optimum answer to the problem contains optimum answer to smaller sub-problems.

CYCLE

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

THINKING

详细地复习了Dynamic Programming。TA认为最大的优势在于代码中不需要建立graph，只需要迭代/递归计算每步的state即可。

个人的想法：optimal sub-structure即为递归的思路，dynamic programming的思路很可能不超过这个范畴。

L20. DYNAMIC PROGRAMMING II

SUMMARY

* DP ≈ "careful brute force"

* DP ≈ guessing + recursion + memoization

* DP ≈ dividing into reasonable # subproblems whose solutions relate — acyclicly — usually via guessing parts of solution.

* time $=$ $\#$ subproblems $\times$ $\underbrace{\textbf{time/subproblem}}_{\substack{\text{treating recursive calls as } O(1) \\ \text{(usually mainly guessing)}}}$

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

* DP ≈ shortest paths in some DAG

5 EASY STEPS TO DYNAMIC PROGRAMMING

1. define subproblems   <u>count # subproblems</u>
2. guess (part of solution)   <u>count # choices</u>
3. relate subproblem solutions   <u>compute time/subproblem</u>
4. recurse $+$ memoize <u>time $=$ time/subproblem $\cdot \#$ subproblems</u>
   OR build DP table bottom-up
   check subproblems acyclic/topological order
5. solve original problem: $=$ a subproblem
   OR by combining subproblem solutions $\Longrightarrow$ <u>extra time</u>

TEXT JUSTIFICATION

an example of constructing a DP algorithm

AUX

Parent Pointer

when you have the min weight, reconstruct the route

Memoization

Recursion to Iteration (DAG)

THINKING

本讲概括了DP应对一般问题的思路，并举例说明。下一讲再看看更多的问题和思路。

总的来说DP适合优化（optimization）问题——找到问题的最大或最小值。一些工具为：

- Parent Pointer（父节点指针）：用来重构路径
- Memoization：使得每个子问题都只计算一次
- DAG：从递归构建为迭代，从而节约计算资源

R20. DYNAMIC PROGRAMMING: BLACKJACK

重复了讲座的内容，没有新的东西，，

LECTURE 21: DYNAMIC PROGRAMMING III

DEFINING SUBPROBLEMS

* problems from L20 (text justification, Blackjack) are on sequences (words, cards)
* useful problems for strings/sequences x:

- $\Theta(n)$
  - suffixes $x[i:]$
  - prefixes $x[:i]$
- $\Theta(n^2)$
  - substrings $x[i:j]$

PARENTHESIZATION

2. guessing = outermost multiplication $(\underbrace{\ldots}_{k-1})(\underbrace{\ldots}_{k})$
   - $\implies \#$ choices $= O(n)$
3. subproblems = prefixes & suffixes? NO

   = cost of substring $A[i:j]$
   - $\implies \#$ subproblems $= \Theta\left(n^2\right)$
4. recurrence:
   - $\mathrm{DP}[i,j] = \min(\mathrm{DP}[i,k] + \mathrm{DP}[k,j] +$ cost of multiplying $(A[i] \cdots A[k-1])$ by $(A[k] \cdots A[j-1])$ for k in range $(i+1, j))$
   - $\mathrm{DP}[i, i+1] = 0$

   $\implies$ cost per subproblem $= O(j-i) = O(n)$
5. topological order: increasing substring size. Total time = $O(n^3)$
6. original problem = $DP[0,n]$

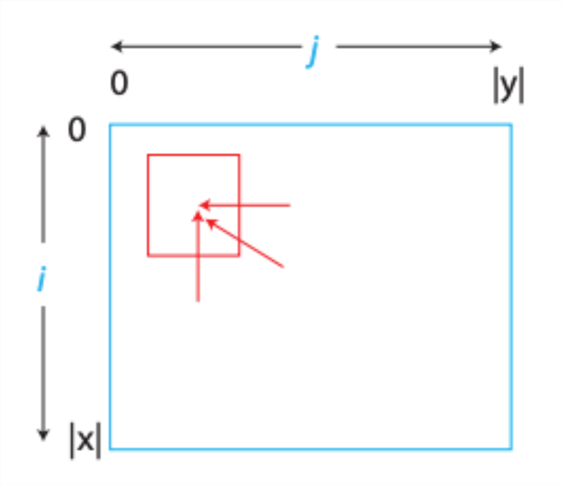
NOTE: Above DP is not shortest paths in the subproblem DAG! **Two dependencies ==⇒ not path!**

EDIT DISTANCE

1. subproblems: $c(i,j) =$ edit-distance $(x[i:], y[j:])$ for $0 \le i < |x|, 0 \le j < |y|$

   $\implies \Theta(|x| \cdot |y|)$ subproblems
2. guess whether, to turn x into y, $(3$ choices $)$:
   - $x[i]$ deleted
   - $y[j]$ inserted
   - $x[i]$ replaced by $y[j]$
3. recurrence: $c(i,j) =$ maximum of:

- cost(delete $x[i]$) $+ c(i+1, j)$ if $i < |x|$,
- cost(insert $y[j]$) $+ c(i, j+1)$ if $j < |y|$,
- cost (replace $x[i] \to y[j]$) $+ c(i+1, j+1)$ if $i < |x| \& j < |y|$

4. topological order:
   - 

   - bottom-up OR right to left
   - only need to keep last 2 rows/columns

KNAPSACK

1. subproblem = value for suffix i:

given knapsack of size X (<= S)

==⇒ # subproblems = O(nS)

Polynomial time

Polynomial time = polynomial in input size

- here $\Theta(n)$ if number S fits in a word
- $O(n \lg S)$ in general
- S is exponential in $\lg S$ (not polynomial)

Pseudopolynomial Time

THINKING

本讲首先介绍了如何定义DP中的子问题，对于序列输入有三种可能性，前序、后序、子序，前二者是线性的，而子序是二次方的。

然后使用DP设计的五步法介绍了几个子序问题：

1. Parenthesization：对于矩阵序列乘积，如何加括号决定相乘的顺序。
    1. 这里的关键是guessing，即最外层相乘发生的位置
    2. 因为需要知道所有的可能的子序的乘积，因此子问题的数量是二次方的
2. Edit Distance
    1. 因为包含删除和置入，因此两个序列的子问题（后序）的位置可能不同，因此两个序列的后序的组合的数量就是二次方的
    2. 对于所有子问题的处理都只有三种操作，因此是常数的
3. Knapsack：旅行背包如何填满后最有用，总空间为S
    1. 由于不知道处理第 i 项的时候背包还有多少空余，因此子问题为用有X空间时决定是否放入第 i 项，因此数量为NS
    2. guess即为放入还是不放入第 i 项

R21. DYNAMIC PROGRAMMING: KNAPSACK PROBLEM

THE KNAPSACK PROBLEM

DAG Shortest-Path Solution

a graph of # nS nodes in order
each node is an item with remained space/weight
each edge is the value gained (negative: shortest path algorithm)
each node pointed to two sub-item nodes: take the item or not

find shortest path in a DAG (one-step Bellman-Ford)

Dynamic Programming Solution

- # nS subproblems:
  $$dp[i][j] = \max \begin{pmatrix} dp[i+1][j] & \\ dp[i+1]\left[j - s_i\right] + v_i & \text{if } j \geq s_i \end{pmatrix}$$
- 2 guesses: take or not
- topological order: $\{n, n-1 \ldots 0\}$

```
KNAPSACK(n, S, s, v)
  for i in {n, n − 1 . . . 0}
    for j in {0, 1 . . . S}
      if i == n
        dp[i][j] = 0 // initial condition
      else
        choices = []
        APPEND(choices, dp[i + 1][j])
        if j ≥ si
          APPEND(choices, dp[i + 1][j − si] + vi)
        dp[i][j] = MAX(choices)
  return dp[0][S]
```

POLYNOMIAL TIME VS PSEUDO-POLYNOMIAL TIME

The amounts of time required to solve some worst-case inputs to the Knapsack problem:

Metric	Baseline	Double n	Double b
n	100 items	200 items	100 items
b	32 bits	32 bits	64 bits
Input size	3,200 bits	6,400 bits	6,400 bits
Worst-case S	$2^{32} - 1 = 4 \cdot 10^9$	$4 \cdot 10^9$	$2^{64} - 1 = 1.6 \cdot 10^{19}$
Running time	$4 \cdot 10^{11}$ ops	$8 \cdot 10^{11}$ ops	$1.6 \cdot 10^{21}$ ops
Input size	1x	2x	2x
Time	1x	2x	$4 \cdot 10^9$x

THINKING

这节复习课用Knapsak（最大价值背包填充）问题介绍了Shortest Path问题基于图（graph）的思路和基于动态规划（dynamic programming）的思路，是很好的复习乃至总结，可以更好地建立图和DP之间的关系的直觉。

有一个想法：DP可以解决的问题是一类特定的图问题，即递归问题，子问题也是一个完整的母问题的问题。

L22. DP IV: GUITAR FINGERING, TETRIS, SUPER MARIO BROS.

2 KINDS OF GUESSING

(A) In (3), guess which other subproblems to use (used by every DP except Fibonacci)

(B) In (1), create more subproblems to guess/remember more structure of solution used by knapsack DP

- effectively report many solutions to subproblem.
- lets parent subproblem know features of solution.

PIANO/GUITAR FINGERING

Piano

1. subproblem = min difficulty for suffix notes $[i:]$ given finger f on first $\operatorname{note}[i]$ (**<u>create more subproblems to guess</u>**)
   - $n \cdot F$ subproblems
2. guessing = finger g for next note $[i+1]$
   - $\Longrightarrow F$ choices
3. recurrence:
   - $DP[i, f]=\min (DP[i+1, g]+d($ note $[i], f$, note $[i+1], g)$ for g in range $(F))$

```
$\operatorname{DP}[n, f]=0$
$\Longrightarrow \Theta(F)$ time/subproblem
```

4. topo. order:
   - for i in reversed $(\operatorname{range}(n))$:
   - for f in $1,2, \ldots, F$:
   - total time $O\left(n F^{2}\right)$
5. orig. prob. = $\min (\operatorname{DP}[0, f]$ for f in $1, \ldots, F)$
   - (guessing very first finger)

Guitar

Up to S ways to play same note! (where S is # strings)

- redefine "finger" = finger playing note + string playing note
- $\Longrightarrow F \rightarrow F \cdot S$

Generalization

Multiple notes at once e.g. chords

- input: notes $[\mathrm{i}]=$ list of $\leq F$ notes (can't play >1 note with a finger)
- state we need to know about "past" now assignment of F fingers to $\leq F+1$ notes / null $\Longrightarrow(F+1)^F$ such mappings

THINKING

本讲介绍了DP的第二种guessing（猜测）方式：通过条件增加条件设置更多的sub-problem（除了prefix、suffix、subfix以外）。而这些条件是能够进行递归的必要条件，如：

- 在kanpsack中，如果子问题不包含背包剩余多少空间/重量，则无法通过猜测当前问题（是否拿该物品）来定位到子问题：子问题不包含剩余空间/重量，因此和是否拿该物品无关
- 在fingering中，如果子问题不包含第一个音符使用哪个手指，则无法通过猜测当前问题（使用哪个手指弹该音符）确定到子问题的权重

第一个的类型是子问题和母问题的猜测无关，第二个的类型是无法确定母问题到子问题的权重

R22. DYNAMIC PROGRAMMING: DANCE DANCE REVOLUTION

Goal

- hit all the notes
- minimize efforts
  - maximize appearance
  - minimize possibility of failure
  - maximize possibility of win: multiply all possibilities
    - $\implies \log()$ to **transform multiplication to addition**

THINKING

这节复习课最有价值的地方是开始介绍的将乘法转换为加法的方法（使用log），从而可以使用shortest path的算法。当然，对于乘法还是可以使用DP的。

之后的内容和讲座中的instrument fingering是重复的，而且大量讨论集中在对于问题的解读而非算法上。

PSET 7

需要用到PIL，因此把需要用到的python2的代码转换为了3的