# CS 170 DISCUSSION 8

## DYNAMIC PROGRAMMING

Raymond Chan
raychan3.github.io/cs170/fa17.html
UC Berkeley Fall 17

# DYNAMIC PROGRAMMING

- Recursive problems uses the subproblem(s) solve the current one.
- Dynamic programming recognizes that such problems have many and overlapping subproblems.
- Can have one or more starting states.

# DYNAMIC PROGRAMMING

- Define subproblems
  - ex. S(i, j) is some solution of subproblem (i, j) of n
- Recurrence relation of subproblems
  - How S(i, j) can use subproblems to solve it
- Base Case(s)
  - ex. S(i, j) = 0 when i = j
- Most have polynomial complexity. Some are exponential.
- Better running time than backtracking, brute-force.

# DYNAMIC PROGRAMMING

- Tree Recursion Memoization
  - Recognize that tree recursive calls overlap.
  - Cache return values for subproblems so that you can use later.
  - Top Down Approach.
- Iterative Dynamic Programming
  - Starts from base case.
  - Expand subproblems until you get to the subproblem you want.
  - Recognizes which subproblems comes first.
  - Bottom Up Approach.

# DYNAMIC PROGRAMMING

- Directed Acyclic Graph underlying structure
- Each subproblem is a vertex.
- Directed edges (u, v) represents constraint that we need to solve subproblem u before subproblem v.

# LONGEST INCREASING SUBSEQUENCE

- Problem
  - Given sequence of number $a_{1,}$ $a_{2, \ldots,}$ $a_n$, find longest increasing sequence of numbers.
- Subproblem
  - L(i) as longest increasing sequence up to the i-th number.
  - Having a partial solution for your original problem.
  - L(i) needs longest increasing sequence of j, for j < i.
  - Goal is L(n)

# EXAMPLES

- Longest Increasing Subsequence
- Edit Distance
- Knapsack with and without repetition
- note: may not cover any (if at all) of the examples.

# LONGEST INCREASING SUBSEQUENCE

- Base Case
  - $L(i) = 1$
- Recurrence
  - $L(i) = \max(\ L(i), L(j) + 1)\ )$ for all $j < i$ and $a[j] < a[i]$
  - Either we start new sequence with base case, or add $a_i$ to longest prior sequence given still increasing

# LONGEST INCREASING SUBSEQUENCE

```
for i = 1..n
  L(i) = 1
  for j = 1..i - 1
    if a[j] < a[i]:
      L(i) = max(L(i), L(j) + 1)
```

Add pointer to actually find sequence.

```
for i = 1..n
  L(i) = 1
  prev(i) = i
  for j = 1..i - 1
    if a[j] < a[i]:
      L(i) = max(L(i), L(j) + 1)
      if updated:
        prev(i) = j
```

Time complexity: $O(n^2)$

Space complexity: $O(n)$

Can also think of it as longest DAG of increasing subsequences.

# EDIT DISTANCE

- Problem
  - Given 2 strings x[1..n] and y[1..m], find <u>edit distance</u> between the 2.
- Subproblem
  - Look at only partial strings.
  - x[1..i] and y[1..j].
  - E(i, j) is the edit distance of x (up to index i) and (y up to index j).
  - Answer at E(n, m).

# EDIT DISTANCE

- Recurrence Relation
  - $E(i, j) = \min(E(i - 1, j) + 1, E(i, j - 1) + 1, E(i - 1, j - 1) + \text{diff}(i, j))$
    - $\text{diff}(i, j) = 1$ if $x[i] \neq x[j]$ else $0$
    - Can delete $x[i]$ and get 1 plus edit distance of all characters before $i$ and at current $j$.
    - Can insert $y[j]$ and get 1 plus edit distance of all characters at current $j$ and before $i$.
    - If $x[i] = y[j]$, then we have edit distance of character before $i$ and $j$.
    - If $x[i] \neq y[j]$, then we substitute $y[j]$ for $x[i]$. 1 plus above.

# EDIT DISTANCE

- Base cases
  - $E(i, 0) = i$, $E(0, j) = j$
- Time complexity: $O(nm)$
- Space complexity: $O(nm)$

# EDIT DISTANCE

- If stuck, try drawing a 2D matrix.
- Number of arguments in subproblem definition determines dimension of matrix.

| | j | | P | O | L | Y | N | O | M | I | A | L |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| i | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| | 0 | | | | | | | | | | | |
| E | 1 | | | | | | | | | | | |
| X | 2 | | | | | | | | | | | |
| P | 3 | | | | | | | | | | | |
| O | 4 | | | | | | | | | | | |
| N | 5 | | | | | | | | | | | |
| E | 6 | | | | | | | | | | | |
| N | 7 | | | | | | | | | | | |
| T | 8 | | | | | | | | | | | |
| I | 9 | | | | | | | | | | | |
| A | 10 | | | | | | | | | | | |
| L | 11 | | | | | | | | | | | **ANS** |

# KNAPSACK WITH REPETITION

- Problem
  - Given items with values and weights, find the highest value items such that total weight is at most W. Can re-use items (multiset).
- Subproblem
  - At some point, you have a knapsack of items and weight.
  - K(w) as best value knapsack with weight w.
  - Need solutions to K(w') for w' < w.
  - Answer at K(W)

# KNAPSACK WITH REPETITION

- Recurrence Formula
  - $K(w) = \max_i ( K(w - w_i) + v_i )$ for $w - w_i \geq 0$
  - Look at item i. If we can still fit in bag, put it in to see if can get better value for weight w with value of this item.
- Base Case
  - $K(0) = 0$
- Time complexity: $O(nW)$
- Space complexity: $O(W)$

# KNAPSACK WITHOUT REPETITION

- Problem
  - Given items with values and weights, find the highest value items such that total weight is at most W. Cannot re-use items (subset).
- Subproblem
  - At some point, you have a knapsack of items and weight.
  - But lets say we have only look at items 1..i out of n items.
  - K(w, i) as best value knapsacks with weight w having only considered items 1..i.
  - Find K(W, n).

# KNAPSACK WITHOUT REPETITION

- Recurrence Relation
  - Look at item i. Either we add it to our bag (if it fits) or we don't.
  - $K(w, i) = \max ( K(w - w_i, i - 1) + v_i, K(w, i - 1) )$
  - Typical tree recursion choice at each step.
- Base Case
  - $K(0, 0) = 0$
- Time Complexity: $O(nW)$
- Space Complexity: $O(nW)$

# KNAPSACK TIME COMPLEXITY

- Knapsack runtime: O(nW).
- Length of an array is the determining factor for most inputs.
  - Processors can handle those 4-8 byte data types easily.
  - Adding one more elements adds 4-8 bytes.
- In knapsack, I can easily have you find best value bag with weight that is very large (e.x. 1,000,000,000).
- W is represented as binary. Add 1 more bit to W, double the running time.
- Looking at the "size" of input (what number is W?), we have polynomial.
- But when I look at the length of input (number of bits in W), we have exponential.