Dynamic Programming Examples

Sam McCauley

April 1, 2024

Welcome Back!

- Two weeks is surprisingly short!
- Assignment 4 due Wednesday
  - Individual assignment
  - Only uses material from before Spring Break
- Assignment 5 out Wednesday as well
  - Group assignment; last assignment before midterm
  - Probably I'll post a short, optional assignment the next week
- Today: start with something familiar, then extend to new things
- Questions?

Longest Increasing Subsequence

Longest Increasing Subsequence

- Given: an arbitrary array A of length n
- Goal: find the length of the longest subsequence of elements that are in sorted order

1	2	10	3	7	6	4	8	11	3	1

Longest Increasing Subsequence

- Given: an arbitrary array A of length n
- Goal: find the length of the longest subsequence of elements that are in sorted order

1	2	10	3	7	6	4	8	11	3	1

The longest increasing subsequence has length 6.

LISE Using Dynamic Programming

Subproblem: $L[i]$ stores the longest increasing sequence ending at $A[i]$

- Base Case: $L[0] = 1$
- How to Fill in $L[i]$: First, create a set M consisting of all entries in A that are:
    - before i in A, and
    - less than $A[i]$
- $L[i] = 1 + \max_{m \in M} L[m]$
- Running time: $O(n^2)$
- How to find the solution: LIS = $\max_j L[j]$

LIS Using Dynamic Programming

- First set $L[0] = 1$
- Fill out each $L[i]$ by finding previous elemements smaller than i and taking the max
- Take the max $L[i]$ after we are done to find the LIS
- Takes $\Theta(i)$ time to fill out $L[i]$, giving $\Theta(n^2)$ time overall.

1	2	10	3	7	6	4	8	11	3	1

New Ideas for LIS

What did we leave unsolved?

- We gave a method to find the *length* of the LIS. What if I want the actual elements?

- I promised that we can do better than $O(n^2)$. It's possible to get to $O(n \log n)$ using some clever bookkeeping.
    - The recursion is the same! We just store extra information to allow us to use a binary search rather than a linear scan to take the max
    - We won't go over this in this class—I'd rather focus on key DP principles rather than a nontrivial technique to speed it up in one particular cae

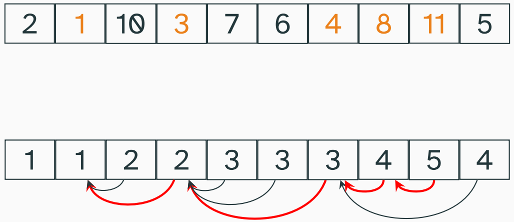
Recovering the LIS Solution

1	2	10	3	7	6	4	8	11	3	1

- Recall: our solution *cost* was $L[i] = 1 + \max_{m \in M} L[m]$; M consists of entries $L[j]$ with $j < i$ and $L[j] < L[i]$
- What elements are in the LISE of $A[i]$ (the longest increasing subsequence that must include $A[i]$?
    - $A[i]$ is! And?
    - All the elements in the LISE of $A[m]$ (where m is the max above)
    - What do we need to store to get the solution back?
        - Store the "m" for each element! Can just store them in an array
        - Doesn't matter how we break ties
        - Store -1 if there is no m (i.e. if M is empty)

Recovering the LIS Solution

Visually:



Recovering the LIS Solution

What we actually store:

Original array *A*:

2	1	10	3	7	6	4	8	11	5

Dynamic Programming array *L*:

1	1	2	2	3	3	3	4	5	4

Solution array *B* storing *m* values:

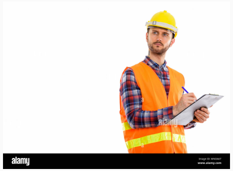
-1	-1	1	1	3	3	3	6	7	6

Recovering the LIS Solution

```
$i =$ max value in $L$
$S = \emptyset$ // holds our solution
while $i \neq -1$:
    add $i$ to $S$
    $i = B[i]$
```

- It took $O(n^2)$ time to fill out L and B
- How much time does it take to find the solution S using the above?
    - $O(n)$
- Total time: $O(n^2)$ to find the LIS!

Finding DP Solutions



- Dynamic programming: use the solution to already-solved subproblems to find solutions to a larger subproblem (a.k.a. recursion)

- To keep track of the solution: write down what subproblems we used to find the new solution

- By backtracking through what subproblems were used for the optimal cost, we can find the actual solution

Edit Distance

Knapsack

A familiar problem?

A familiar problem?

A familiar problem?

Packing is *Hard*

- Sometimes: you pack a suitcase, dishwasher, backpack, etc.
- Items don’t fit
- You take everything out and put it back in and suddenly it fits
- Can we come up with an algorithm to pack items efficiently? Can we beat brute force?

Today: Weight limit only

Knapsack

- You are packing a bag, with a weight capacity C
- You have a collection of items to put in your bag
- Each item i has a weight w_i and a value v_i (both nonnegative integers)
- Choose a subset of items with *total weight* at most C
- Goal: maximize the *total value* of the items you pack

Knapsack

- Does greedy work? How could we greedily pack a bag?
- Option 1: pick the highest-value item. Counterexample? [On Board #1]
- Option 2: pick the lowest-weight item. Counterexample?
- Option 3: pick the item maximizing value/weight. Counterexample?

Recursive Knapsack

- Goal for the next portion of class: come up with the dynamic program for knapsack together [On Board #2]
- There are likely to be some false starts! I’m not writing the solution line by line.
- (Also there are some ideas that don’t work that I specifically want to discuss :) so we may circle back to some suggestions)