Dynamic Programming Examples

Sam McCauley

April 7, 2025

Welcome Back!

- Two weeks is surprisingly short!
- Problem Set 5 due Wednesday
- Today: start with something familiar, then extend to new things
- Last lecture before break is recorded and posted
- 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.

Breaking down a DP

- What does an optimal solution look like? Can we break it down further?

- Strategy: split into several cases
  - The optimal solution must satisfy one of these case
  - We don’t know which yet! That’s OK
  - Get the cost of each case recursively and take whichever has lowest cost

Longest Increasing Subsequence: Cases

- 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

We solve a slightly different problem: longest increasing subsequence ending at the last element. Let's look at 11.

The *second-to-last* element must have been one of 1, 2, 10, 3, 7, 6, 4, 8.

Strategy: find the cost of the longest increasing subsequence ending at each. Take the smallest!

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

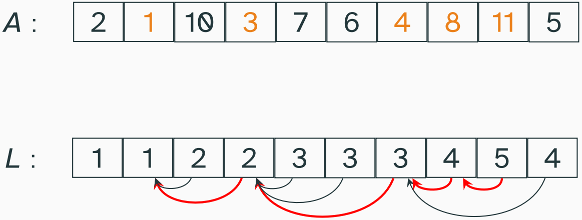
Recovering the LIS Solution

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 best value of m for each i (or -1 if M empty):

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

Can fill in B while filling in L!

Recovering the LIS Solution

```
i = max value in L
S = ∅ // holds our solution
while i ≠ −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? [Blackboard]
- 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 [Blackboard]
- 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)

Recursive Knapsack Solution

- Subproblem: (i, c): what is the largest-value solution among the first i items with total weight at most c?
- Memoization structure: $n \times C$ matrix (storing $OPT(i, c)$ for $i \in \{1, \ldots, n\}$ and $c \in \{1, \ldots, C\}$.
- Recurrence: $OPT(i, c) = \max\{OPT(i-1, c), v_i + OPT(i-1, c-w_i)\}$
- Final answer: $OPT(n, C)$
- Before moving forward: what subproblems do we need to solve in order to fill in $OPT(i, c)$?
  - In what order should we fill out the table?
  - Base cases?
  - Answer: we need all entries in $OPT(i-1, c)$ to fill out any entry in $OPT(i, c)$. So go item by item. Our base case must fill out all entries in $OPT(1, c)$.

Recursive Knapsack Solution

- (recall) Memoization structure: $n \times C$ matrix (storing $OPT(i, c)$ for $i \in \{1, \ldots, n\}$ and $c \in \{1, \ldots, C\}$).
- Evaluation order: Row-major order (row by row: fill in $OPT(i, c)$ for $c \in \{1, \ldots, C\}$ before filling in $OPT(i + 1, c)$ for $c \in \{1, \ldots, C\}$).
- Base cases: $OPT(1, c) = v_1$ if $c \geq w_1$, $OPT(1, c) = 0$ if $c < w_1$.
- Space: $O(nC)$ Time: $O(nC)$

A Comment on Running Time

- Running time is $O(nC)$
- In algorithms we generally want a "polynomial" running time (i.e. a polynomial in the *size* of the input). All running times we've seen so far in this class were polynomial.
- Is this polynomial in the size of the input?
  - No! The size of the input is $O(n + \log_2 C)$ (it takes $\log_2 C$ bits to write C down)
  - C is exponential in $\log_2 C$. So this running time is not polynomial
- This knapsack DP is pseudopolynomial: the running time is polynomial in the *value* of the input, not the *size*

Pseudopolynomial Running Time Comments

- When is pseudopolynomial running time a big downside?
- Is this a practical problem?
- What happens when the weights of the items are not integers? Does our DP work? Can we make it work?