Dynamic Programming - Knapsack

Key ideas of dynamic programming: identify subproblems (not too many) and an order of solving them such that each subproblem can be solved by combining previously solved subproblems.

Recall the knapsack problem: Given items 1, 2, \ldots, n, where item i has weight $w_i$ and value $v_i$ \((w_i, v_i \in \mathbb{Z})\) choose a subset $S$ of items s.t. $\sum_{i \in S} w_i \leq W$, capacity of knapsack and $\sum_{i \in S} v_i$ is maximized.

Recall that we considered the fractional version (can use fractions of items e.g. flour, rice) where greedy alg. works.

Today we consider the 0-1 version where items are indivisible (e.g. flashlight, tent).

First attempts: Like weighted interval scheduling, distinguish whether item n is in or out:

- If $n \notin S$ - look for opt. soln. for 1 \ldots n-1
- If $n \in S$ - want subset $S$ of 1 \ldots n-1 with $\sum_{i \in S} w_i \leq W - w_n$ the space left in the knapsack

As for coin changing problem, we need different subproblems for different knapsack capacities.
Subproblems are one for each pair \( i, w \), \( i = 0 \ldots n \), \( w = 0 \ldots W \).

Find subset \( S \subseteq \{1 \ldots i\} \) s.t.

\[
\sum_{i \in S} w_i \leq w \quad \text{and} \quad \sum_{i \in S} v_i \text{ is maximized}
\]

Let \( M(i, w) = \max_{\{i \in S\}} \sum_{i \in S} v_i \)

To find \( M(i, w) \)

* if \( w_i > w \) then \( M(i, w) \leftarrow M(i-1, w) \)
* else \( M(i, w) \leftarrow \max \{ M(i-1, w) \ \text{or} \ \sum_{i \in S} v_i + M(i-1, w-w_i) \} \) \( \checkmark \) use \( i \)

Pseudocode and ordering of subproblems:

```plaintext
Use matrix \( M[0 \ldots n, 0 \ldots W] \)
Initialize \( M[0, w] \leftarrow 0 \), \( w = 0 \ldots W \)
for \( i = 1 \ldots n \)
for \( w = 0 \ldots W \)
compute \( M[i, w] \) using \( \checkmark \)
```

Analysis:

We have a nested loop \( n \cdot W \cdot c \) constant work for \( \checkmark \) loop for \( i \)
So \( O(n \cdot W) \)

This is not a polynomial time algorithm. It is pseudo-polynomial time.

The input is \( w_1 \ldots w_n, v_1 \ldots v_n, W \)
Size of input is sum of \# bits.
W is one of the numbers in the input. The size of the input counts the size of W — let’s say it has k bits. \( k = \Theta(\log W) \)

But the algorithm takes \( O(n \cdot W) \) — that is \( O(n \cdot 2^k) \) so it’s exponential in the input size.

Run-time is polynomial in the value of W rather than size of W.

Finding the actual solution for knapsack. Two methods:

1. backtracking
2. store solution with \( M \) (after original code)

1. Backtracking: Use \( M \) to recover solution
   
   \[
   i \leftarrow n, \quad w \leftarrow W \\
   \text{while } i > 0 \quad \\
   \quad \text{if } M(i, w) = M(i-1, w) \quad \text{* didn’t use } i \\
   \quad \quad i \leftarrow i - 1 \\
   \quad \text{else} \quad \quad \text{* used } i \\
   \quad \quad \text{output } i \\
   \quad \quad i \leftarrow i - 1, \quad w \leftarrow w - w_i 
   \]

Time: \( O(n) \)
2. enhance original code
   when we set \( M(i, w) \)
   also set \( \text{Flag}(i, w) \) - do we use item \( i \) or not to get \( M(i, w) \)
   (we still need backtracking)
   or even store \( \text{Soln}(i, w) \) - list of items to get \( M(i, w) \)
   (no backtracking needed)

Trade-offs: (2) uses more space
   (1) duplicates tests used to compute \( M \)

A simpler related problem
(relevant when we study NP-hardness)
Subset Sum. Given \( n \) natural numbers
\( a_1, \ldots, a_n \) and number \( K \), is there a subset \( S \subseteq \{ a_1, \ldots, a_n \} \) s.t.
\( \sum_{i \in S} a_i = K \).

There is a pseudo-polynomial time dynamic programming algorithm
HINT \( M(i, k) \) \( i = 0 \ldots n \), \( k = 0 \ldots K \)
\( = \text{YES/NO} \), is there a subset of \( \{ a_1, \ldots, a_i \} \) adding to \( k \).
Common subproblems in dynamic programming

1. input $x_1 \cdots x_n$
   subproblems $x_1 \cdots x_i$
   $\#$ subproblems $n$

2. input $x_1 \cdots x_n$
   subproblems $x_i \cdots x_j$
   $\#$ subproblems $O(n^2)$

3. input $x_1 \cdots x_n, y_1 \cdots y_m$
   subproblems $x_1 \cdots x_i$ and $y_1 \cdots y_j$
   $\#$ subproblems $O(n \cdot m)$

4. input rooted tree on $n$ nodes
   $\#$ subproblems $O(n)$

5. 0-1 Knapsack and Subset Sum
   with $n \times w$ subproblems
   weight
**Chain Matrix Multiplication**

**Problem.** For matrices $M_1, M_2, \ldots, M_n$, compute $M_1 \cdot M_2 \cdot \ldots \cdot M_n$.

For 2 matrices $C = A \cdot B$, $A - d_1 \times d_2$ and $B - d_2 \times d_3$.

Then $C$ is $d_1 \times d_3$ and computing $D$ takes $d_1 \cdot d_2 \cdot d_3$ scalar multiplication (plus additions),

**Cost** $= d_1 \cdot d_2 \cdot d_3$

What order should we multiply the $M_i$'s in, to minimize cost?

**Example** $A_1 = 2 \times 10, A_2 = 10 \times 1, A_3 = 1 \times 4$

\[
\begin{align*}
\underbrace{(A_1 \cdot A_2)}_{2 \times 10 \times 1} \cdot A_3 & \quad \underbrace{A_1 \cdot (A_2 \cdot A_3)}_{10 \times 1 \times 4} \\
& \quad \underbrace{2 \times 1 \cdot 4}_{2 \times 1 \times 4} \quad \underbrace{2 \times 1 \cdot 4}_{2 \times 1 \times 4} \\
& = 28 \quad = 120
\end{align*}
\]

**Ex:** Find an example where greedy does not work.

Deciding the order to multiply the $M_i$'s

- Parenthesizing the expression $M_1 \ldots M_n$
- Building a binary tree

*E.g.* $(M_1, M_2), (M_3, M_4)$, $((M_1, M_2) \cdot M_3) \cdot M_4$
How many ways are there to:

\[ P_n = \sum_{i=1}^{n} P_i \cdot P_{n-i} \quad \text{where } i \text{ chooses root of tree} \]

\[ P_n = n^{th} \text{ Catalan no.} \quad P_5 = 14 \quad P_{15} = 2,674,940 \]

\[ P_n \in \mathcal{O} \left( \frac{4^n}{n^2} \right) \] — so don’t try them all!

Subproblems — best way to multiply \( M_i \cdots M_j \)

Notation: Let \( M_i \) have dimensions \( d_{i-1} \times d_i \)

so some matrix is \( d_0 \times d_n \)

Let \( C(i,j) = \min \text{ no. scalar multi. to compute } M_i \cdots M_j \)

\[ C(i,i) = 0 \]
\[ C(i,j) = \min_{k = i \cdots j-1} \left\{ C(i,k) + C(k+1,j) + d_{i-1} \cdot d_k \cdot d_j \right\} \]

because:

\[ (M_i \cdots M_k) \cdot (M_{k+1} \cdots M_j) \]
\[ d_{i-1} \times d_k \quad d_k \times d_j \]

Compute \( C(i,j) \)'s in increasing order of \( j-i \)

Computing \( C(i,j) \) takes \( O(n) \) time (try \( n \) values of \( k \))

Total time \( O(n^2 \cdot n) = O(n^3) \)

Total time for each subproblems

Best alg. for this problem is \( O(n \log n) \)
Pseudo-code

for $i = 1 \cdots n$
$C(i, i) \leftarrow 0$
end

for $d = 1 \cdots n$
    for $i = 1 \cdots n - d$
        $j \leftarrow i + d$
        $C(i, j) \leftarrow \infty$
        for $k = i \cdots j - 1$
            temp \leftarrow C(i, k) + C(k+1, j) + (d(i-1) \cdot d(k) \cdot d(j))$
            if $C(i, j) > temp$
                $C(i, j) \leftarrow temp$
                $k(i, j) \leftarrow k$
            end
        end
    end
end

Min cost of opt so$m$ is $M(1, n)$
and we can recover the solution using $k(1, n)$

$(M_1 \cdots M_k) \cdot (M_{k+1} \cdots M_n)$
Memoization

- use recursion, rather than explicitly solving all subproblems bottom-up as we've been doing so far.

- danger - that you solve the same subproblem over and over (possibly taking exponential time, e.g.
  \( T(n) = 2T(n-1) + O(1) \) is exponential)

- fix - when you solve a subproblem, store the solution. Before (re)-solving, check if you have a stored solution. Solutions can be stored in a matrix or in a hash table.

- advantage - maybe you don't solve all subproblems.