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 $1, 2 \cdots n$, 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. $\sum_{i \in S} w_i \le W$ ← 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 attempt: Like weighted interval scheduling, distinguish whether item n is IN or OUT.

if $n \notin S$ — look for opt. soln for $1 \cdots n-1$

if $n \in S$ — want subset S of $1 \cdots n-1$ with

$$\sum_{i \in S} w_i \le \underbrace{W - w_n}_{\text{the space left in the knapsack}}$$

As for coin changing problem, we need different subproblems for different knapsack capacities.

Subproblems: one for each pair i, w, $i = 0 \ldots n$, $w = 0 \ldots W$

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

$$\sum_{i \in S} w_i \le w \quad \text{and} \quad \sum_{i \in S} v_i \text{ is maximized}$$

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

To find $M(i, w)$

* - if $w_i > w$ then $M(i,w) \leftarrow M(i-1, w)$
- else $M(i,w) \leftarrow \max \begin{cases} M(i-1, w) & \text{/* don't use } i \\ v_i + M(i-1, w-w_i) & \text{/* use } i \end{cases}$

Pseudocode and ordering of subproblems:

Use matrix $M[0..n, 0..W]$

initialize $M[0, w] \leftarrow 0$ $\quad w = 0 \ldots W$

for $i = 1 \ldots n$

for $w = 0 \ldots W$

compute $M[i, w]$ using *

Analysis:

We have a nested loop $\underset{\text{loop for } i}{n} \cdot \underset{\text{loop for } w}{W} \cdot c$ ← constant work for *

so $O(n \cdot W)$

This is not a polynomial time algorithm. It is <u>pseudo-polynomial</u> 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's $O(n \cdot 2^k)$ so it's exponential in the input size.

Run-time is polynomial in the <u>value</u> of W rather than <u>size</u> of W.

Finding the actual solution for knapsack.
Two methods:

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

1. Backtracking: Use M to recover solution

```
i ← n, w ← W
while i > 0
    if M(i, w) = M(i-1, w)     /* didn't use i
        i ← i-1
    else                       /* used i
        output i
        i ← i-1, w ← w - w_i
end
```

Time: $O(n)$

2. enhance original code
when we set $M(i, w)$
also set $Flag(i,w)$ – do we use item i or not
to get $M(i, w)$

(we still need backtracking)

or even store $Soln(i, w)$ – 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)

<u>Subset Sum</u> Given n natural numbers $a_1 \cdots a_n$ and number K, is there a subset $S \subseteq \{1 \cdots n\}$ s.t. $\sum_{i \in S} a_i = K$.

There is a pseudo-polynomial time dynamic programming algorithm

HINT $M(i, k)$ $i = 0 \dots n$, $k = 0 \dots K$
$= $ YES/NO, is there a subset of $\{1 \cdots i\}$ adding to k.

Common subproblems in dynamic programming

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

   weighted interval scheduling

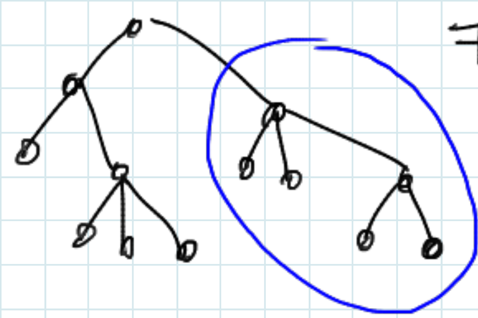
2. input $x_1 \cdots x_n$
   subproblems $x_i \cdots x_j$
   # subproblems $O(n^2)$

   opt. binary search tree

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 $O(n \cdot m)$

   edit distance

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



maybe later

5. 0-1 Knapsack and Subset Sum
   with $n \times W$ subproblems
   weight

Chain Matrix Multiplication

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

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

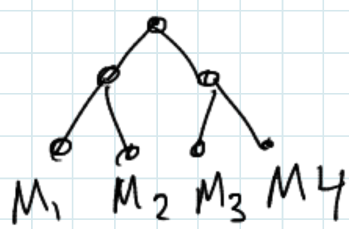
Example $A_1 - 2 \times 10$ $A_2 - 10 \times 1$ $A_3 - 1 \times 4$

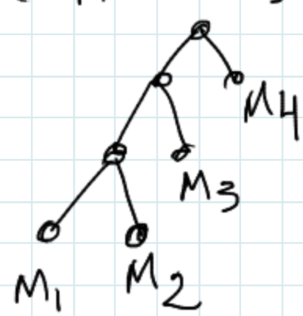
$$\underbrace{\underbrace{(A_1 \cdot A_2)}_{2 \cdot 10 \cdot 1} \cdot A_3}_{2 \cdot 1 \cdot 4} = 28 \qquad \underbrace{A_1 \cdot \underbrace{(A_2 \cdot A_3)}_{10 \cdot 1 \cdot 4}}_{2 \cdot 10 \cdot 4} = 120$$

EX. Find an example where greedy does not work.

Deciding the order to multiply the M_i's
= parenthesizing the expression $M_1 \cdots M_n$
= building a binary tree

e.g. $(M_1 \cdot M_2) \cdot (M_3 \cdot M_4)$ $\qquad ((M_1 \cdot M_2) \cdot M_3) \cdot M_4$





How many ways are there?

$P_n = \sum_{i=1}^{n} P_i \cdot P_{n-i}$ where i chooses root of tree

P_n = nth Catalan no. $P_5 = 14$ $P_{15} = 2,674,440$

$P_n \in \Omega\left(\frac{4^n}{n^2}\right)$ — 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 soln matrix is $d_0 \times d_n$

Let $C(i,j)$ = min. no. scalar mult. to compute $M_i \cdots M_j$

$C(i,i) = 0$

$$C(i,j) = \min_{k=i..j-1} \{ C(i,k) + C(k+1,j) + \underbrace{d_{i-1} \cdot d_k \cdot d_j} \}$$

because:

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

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

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

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

\# subproblems (n^2) — time for each (n)

Best alg. for this problem is $O(n \log n)$

Pseudo-code

```
for i = 1 .. n
   C(i,i) ← 0
end
for diff = 1 .. n
   for i = 1 .. n-diff
      j ← i+diff
      C(i,j) ← ∞
      for k = i .. j-1
         temp ← C(i,k) + C(k+1,j) + d(i-1)·d(k)·d(j)
         if C(i,j) > temp
            C(i,j) ← temp
            k(i,j) ← k
      end
   end
```

Min cost of opt. sol'n 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.

X