Agenda: Dynamic Programming (DP)

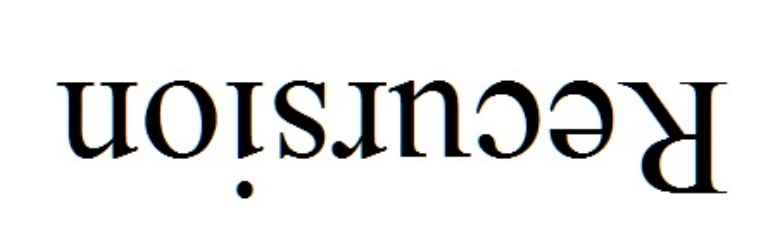
- 0/1 Knapscak (CLRS Exercise 16.2-2; we solve it here)
- Longest Common Subsequence
- Chain Matrix Multiplication

Reading:

- CLRS, Ch. 15: 359-397
- CLRS, Ch. 25: 693-699

Remarks:

- DP is considered one of the most difficult topics of this course
- DP is about recursion upside-down, we will see what it means.



Dynamic programming introduction:

- An algorithm design paradigm
- Usually for optimization problems
- Typically like divide-and-conquer uses solutions to subproblems to solve the problem, BUT
- Key idea: *Avoids re-computation* of repeated subproblems by storing subproblem answers in tables/arrays

General steps in designing a dynamic programming solution:

1. Find a recurrence relation for the problem.

2. Check to see if the recurrence repeatedly makes the same calls, and if all of the recursive calls live in a moderately sized universe.
   In Fibonacci, we make a lot of calls, but only to n possible values: $F(0), F(1), F(2), \ldots, F(n-1)$.

3. Describe an array of values that you want to compute. Each cell in the array will be the result of a possible recursive call, and the value of the solution for the appropriate subproblem.
   $A[i]$ stores the value $F(i)$.

4. Fill the array bottom-up: from the cells corresponding to the base case of the recursion, to cells we now can compute.
   First fill $A[0], A[1]$, then $A[2]$, then $A[3]$,..., then $A[n]$.

5. Extract the solution from the array.
   $A[n]$.

Integral Knapsack

- Recall, we broke into a jewelry store and we are looking to fill our knapsack with the most profitable set of items.
- We have a knapsack with capacity W
- n items with weights $w_1, \ldots, w_n \in \mathbb{N}$ and values $v_1, \ldots, v_n \in \mathbb{N}$. and we want to fill the knapsack without exceeding its capacity.
- Integral: we will have to take an item as a whole, or not take it; we cannot break it
- First attempt; define the problem solution by brute force - Try all possible subsets of items and select the best.

$$OPT\Big(W; w_1, ..., w_n; v_1, ..., v_n\Big) = \max_{\substack{S \subset \{1,...,n\} \\ \text{with } \sum\limits_{i \in S} w_i \leq W}} \{\sum_{i \in S} v_i\}$$

.

- Example: $w_1 = 10,\ w_2 = 10,\ w_3 = 11$
  $v_1 = 10,\ v_2 = 10,\ v_3 = 12,\ W = 20.$

2^{nd} solution: Recursion

- First, determine what to do with the last item - either I take it, gain v_n and recurse on remaining items with capacity $W - w_n$, or I leave it, and recurse on remaining items with capacity W

$$
\begin{aligned}
&OPT\Big(W; w_1, ..., w_n; v_1, ..., v_n\Big) \\
&= \max\Big\{OPT\big(W; w_1, ...w_{n-1}; v_1, ...v_{n-1}\big), \\
&\qquad\qquad v_n + OPT\big(W - w_n; w_1, ...w_{n-1}; v_1, ...v_{n-1}\big)\Big\}
\end{aligned}
$$

- Now, the algorithm makes (at least) 2^n recursive calls — since for each item we try both options (take it or leave it)
- Recursions live in a small domain - a capacity $D \in [0, W]$ and a *prefix of items* $\{1, ..., i\}$
- Overall $(n+1) \times (W+1)$ choices. What does this imply?
- **Tip**: This is a common practice in DP: what to do with the last element: take it, leave it, or do something about it.

3^{rd} solution: use Dynamic programming.

- Step 1: Define array $A[i, D]$, $0 \leq i \leq n$ and $0 \leq D \leq W$ where $A[i, D]$ stores the result of a possible recursion call.
  - I.e., $A[i, D]$ is the value of best possible knapsack of weight at most D using only items $\{1, 2, ..., i\}$.
  - In other words, $A[i, D]$ stores the value of the optimal solution for the subproblem on fewer items (the first i items) and a smaller knapsack (capacity $D \leq W$).
  - The optimal solution's value: $A[n, W]$.

A Side Note: The Integral Knapsack problem has *Optimal Substructure*: an optimal solution can be constructed from optimal solutions of its subproblems. Unlike a greedy algorithm, here there is no guarantee that each choice is optimal, e.g., we either take the last item or not.

3^{rd} solution: use Dynamic programming.

- Step 2: Fill the entries $A[i, D]$ in the array — bottom-up!
  - If $i = 0$ or $D = 0$ then trivially $A[i, D] = 0$.
  - Else, consider item i:
    - If we do not choose item i: knapsack must be packed optimally with items from $1 \ldots (i-1)$.
    - If we choose item i (assuming $D \geq w_i$): $D - w_i$ remaining cap. must be packed with items $1 \ldots (i-1)$.
- So $A[i, D] = \max \begin{cases} A[i-1, D] \\ \text{(if } D \geq w_i\text{)} \ v_i + A[i-1, D - w_i] \end{cases}$
- Filling the array row-by-row (first $i = 1$, then $i = 2$, then $i = 3$,...,till $i = n$), when we reach the $[i, D]$-cell, both $[i-1, D]$ and $[i-1, D-w]$ are filled.

3^{rd} solution: use Dynamic programming.

- $A[i, D] = \max \begin{cases} A[i-1, D] \\ (\text{if } D \geq w_i)\ v_i + A[i-1, D-w_i] \end{cases}$
- Step 3:

  <u>procedure Knapsack$(W, w_1, \ldots, w_n, v_1, \ldots, v_n)$</u>

  for $i \leftarrow 1$ to n do
      $A[i, 0] \leftarrow 0$
  for $D \leftarrow 0$ to W do
      $A[0, D] \leftarrow 0$
  for $i \leftarrow 1$ to n do
      for $D \leftarrow 1$ to W do
          $A[i, D] \leftarrow A[i-1, D]$
          if $(D \geq w_i$ and $A[i, D] < A[i-1, D-w_i] + v_i)$ then
              $A[i, D] \leftarrow A[i-1, D-w_i] + v_i$
  return $A[n, W]$
- Runtime? $O(nW)$

W 1 5 11 5

Finding the Solution

- Step 4: How to find the **set** of items of the optimal packing?
- Consider item n. It can be seen that if $A[n, W] = A[n-1, W]$ then
- n is not in the optimal solution. Else it is in the solution and $A[n, W]$ is obtained from $A[n-1, W-w_n]$ by adding item n.
- Now continue with either $A[n-1, W]$ or $A[n-1, W-w_n]$ to find which of the remaining items is in the optimal solution.
- This suggests the following algorithm:

- procedure Print-Opt-Knapsack (A, i, D)

```
if (i > 0 or D > 0) then
   if (A[i, D] = A[i − 1, D]) then
      Print-Opt-Knapsack (i − 1, D)
   else
      Print-Opt-Knapsack (i − 1, D − w_i)
      Print(i)
```

- Print-Opt-Knapsack(A, n, W) recurrence relation: $T(n) = O(1) + T(n-1)$
- So printing takes $O(n)$ time. Overall runtime is $O(nW)$.

```
 Example:  n = 3 (# of items),  W = 5,
           Items (weight, value): (4,6), (2,5), (2,4)
A[i,D] array:
```

i\D	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0			
2						
3						

Conditions in the inner for loop:

$D \geq w_i$ (item i CAN be in solution)
$A[i, D] < A[i-1, D-w_i] + v_i$ (if holds $A[i, D]$ should be updated)

val wt	0	1	2	3	4	5
	0	0	0	0	0	0
4 2	0	0	4	4	4	4
5 2	0	0	5	5	9	9
6 4	0	0	5	5	9	9

Example: n = 3 (# of items), W = 5,
Items (weight, value): (4,6), (2,5), (2,4)

A[i,D] array:

i\D	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	6	6
2	0	0	5	5	6	6
3	0	0	5	5	9	9

weight	value
4	6
2	5
2	4

Conditions in the inner for loop:

$D \geq w_i$ (item i CAN be in solution)

$A[i, D] < A[i-1, D-w_i] + v_i$ (if holds $A[i, D]$ should be updated)

Longest Common Subsequence (LCS) problem:

- Definitions:
  - Base/letter/character: e.g. a,b,c,d...
  - Sequence/string: $X = x_1, x_2, ..., x_n$ where each x_i is a letter
  - Subsequence: removing zero or more letters from the given sequence
    Note: letters appear in the same order, but not necessarily consecutive
  - Common subsequence of X and Y: a string which is both a subsequence of X and subsequence of Y.
    e.g., *dog* is a common subsequence of *dynamicprogram* and *dough*.

The problem statement:

- LCS problem: given two sequences

  $$X = x_1 x_2 \ldots x_n \text{ and } Y = y_1 y_2 \ldots y_m,$$

  find a maximum-length common subsequence of them.
- Applications:
  - Human (and other species) Genome Project
  - Detecting cheating on HW

LCS (cont'd):

- The LCS problem has the "optimal substructure" ...
  - if x_n is NOT in the LCS — then we only need to compute an LCS of $x_1 x_2 \ldots x_{n-1}$ and $y_1 y_2 \ldots y_m$ ...
  - similarly, if y_m is NOT in the LCS — then we only need to compute an LCS of $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_{m-1}$ ...
  - if x_n and y_m are both in the LCS — then $x_n = y_m$ and we need to compute an LCS of $x_1 x_2 \ldots x_{n-1}$ and $y_1 y_2 \ldots y_{m-1}$ and pad it with x_n.
- So we get the recursion

$$LCS(x_1 ... x_n; y_1 ... y_m) = \max \left\{ \begin{array}{r} LCS(x_1 ... x_{n-1}; y_1 ... y_m), \\ LCS(x_1 ... x_n; y_1 ... y_{m-1}), \\ (\text{if } x_n = y_m)\ 1 + LCS(x_1 ... x_{n-1}; y_1 ... y_{m-1}) \end{array} \right\}$$

- 3 recursive calls on instances size 1 or 2 smaller — blows up to 3^n calls.
- But yet again, they all live in a small domain: first i characters of X, first j characters of Y.

Longest common subsequence (LCS) problem (cont'd):

- Therefore, we define $D[i,j]$ to hold the result of the (i,j)-recursion call: Namely, for each $0 \leq i \leq n$ and $0 \leq j \leq m$, $D[i,j]$ is the length of LCS of $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$.
- The recursive formula tells us how to compute $D[i,j]$:

$$D[i,j] = \max \begin{cases} D[i-1,j], \\ D[i,j-1], \\ D[i-1,j-1]+1, & \text{if } x_i = y_j \end{cases}$$

- Base cases? If one of the strings is of length 0, the answer is 0, i.e., $D[i,0] = D[0,j] = 0$ for any i,j.
- Our goal is to compute $D[n,m]$ — the length of an LCS of X and Y.
- We fill D in a bottom-up fashion, making sure that whenever we fill in $D[i,j]$ then all 3 cells required by the formula have already been filled: $D[i-1,j], D[i,j-1], D[i-1,j-1]$.

LCS (cont'd)

- $\underline{\texttt{procedure LCS}(X, Y)}$

  $n \leftarrow length[X]$
  $m \leftarrow length[Y]$
  $\texttt{for } (i \leftarrow 1 \texttt{ to } m) \texttt{ do}$
  $\quad D[i, 0] \leftarrow 0$
  $\texttt{for } (j \leftarrow 0 \texttt{ to } n) \texttt{ do}$
  $\quad D[0, j] \leftarrow 0$
  $\texttt{for } (i \leftarrow 1 \texttt{ to } n) \texttt{ do}$
  $\quad \texttt{for } (j \leftarrow 1 \texttt{ to } m) \texttt{ do}$
  $\quad\quad D[i, j] \leftarrow D[i-1, j]$
  $\quad\quad \texttt{if } (D[i, j-1] > D[i, j]) \texttt{ then}$
  $\quad\quad\quad D[i, j] \leftarrow D[i, j-1]$
  $\quad\quad \texttt{if } (x_i = y_j \texttt{ and } D[i-1, j-1] + 1 > D[i, j]) \texttt{ then}$
  $\quad\quad\quad D[i, j] \leftarrow D[i-1, j-1] + 1$
  $\texttt{return } D[n, m]$

- Runtime $\Theta(n + m + n \cdot m)$

Longest common subsequence (LCS) problem (cont'd):

- To return a LCS ... trace back — using recursion
  - If $D[i,j] = D[i-1,j]$ print the solution for the $(i-1,j)$-subproblem
  - If $D[i,j] = D[i,j-1]$ print the solution for the $(i,j-1)$-subproblem
  - If $D[i,j] = D[i-1,j-1]+1$ then print the solution for the $(i-1,j-1)$-subproblem and then print the character x_i (same as y_j)
- procedure PrintLCS(D,i,j,X,Y)

```
if (i > 0 and j > 0)
   if (D[i,j] = 1 + D[i-1,j-1]) then
      PrintLCS(D, i-1, j-1, X, Y)
      Print X_i
   else if (D[i,j] = D[i-1,j])
      PrintLCS(D, i-1, j, X, Y)
   else
      PrintLCS(D, i, j-1, X, Y)
```

- Runtime?
  - In each call we do $O(1)$ work and recurse on an instance where either i or j is smaller by 1 (or both are smaller).

$$T(n+m) = O(1) + T(n+m-1)$$

  - Solves to $O(n+m)$.
- So the total runtime: $\Theta(nm)$

Matrix-Chain Multiplication:

- Input: matrices A_1, A_2, $\ldots$, A_n with dimensions $d_0 \times d_1$, $d_1 \times d_2$, $\ldots$, $d_{n-1} \times d_n$, respectively.
- Output: an order in which matrices should be multiplied such that the product $A_1 \times A_2 \times \ldots \times A_n$ is computed using the minimum number of scalar multiplications.
- Fact: suppose A_1 is a $d_1 \times d_2$ matrix, A_2 is a $d_2 \times d_3$ matrix. Then A_1 and A_2 is multipliable, and $B = A_1 \times A_2$ can be computed using $d_1 \times d_2 \times d_3$ scalar multiplications.
  (Yes, we learnt ways to do it faster, let's disregard those for now. In the general case you will replace $d_1 \times d_2 \times d_3$ with some $f(d_1, d_2, d_3)$.)
- Example: $n = 4$ and $(d_0, d_1, \ldots, d_n) = (5, 2, 6, 4, 3)$
  Possible orders with different number of scalar multiplications:

$((A_1 \times A_2) \times A_3) \times A_4$	$5 \times 2 \times 6 + 5 \times 6 \times 4 + 5 \times 4 \times 3 = 240$
$(A_1 \times (A_2 \times A_3)) \times A_4$	$5 \times 2 \times 4 + 2 \times 6 \times 4 + 5 \times 4 \times 3 = 148$
$(A_1 \times A_2) \times (A_3 \times A_4)$	$5 \times 2 \times 6 + 5 \times 6 \times 3 + 6 \times 4 \times 3 = 222$
$A_1 \times ((A_2 \times A_3) \times A_4)$	$5 \times 2 \times 3 + 2 \times 6 \times 4 + 2 \times 4 \times 3 = 102$
$A_1 \times (A_2 \times (A_3 \times A_4))$	$5 \times 2 \times 3 + 2 \times 6 \times 3 + 6 \times 4 \times 3 = 138$

each subchain is split further

subproblem: $A_i \cdots A_j$ $\quad 1 \le i \le j \le n$

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & \cdots & & b_{24} \\ b_{31} & \cdots & & b_{34} \end{bmatrix}$$

2×3 $\quad$ 3×4

$= 2 \times 4$ matrix

$k \times n$ $\quad$ $n \times m$

$O(k \times n \times m)$

$R[1,n] = \min\left(R[1,i] + R[i+1,n] + d_0 \cdot d_i \cdot d_n\right)$

$1 \le i \le n$

Matrix-chain multiplication: 1^{st} Solution — Recursion:

- A recursive solution:
  - Consider the highest level parenthesis: $(A_1 \ldots A_i)(A_{i+1} \ldots A_n)$.
  - This gives two matrices: one of dimensions $(d_0 \times d_i)$ and another of dimension $(d_i \times d_n)$. Multiplying these two requires $O(d_0 d_i d_n)$-time.
  - ...and we still need to find the least-costly way to do the multiplications of the first i matrices and the latter $n-i$ matrices.
  - There are $n-1$ possiblities: i can be anywhere between 1 to $n-1$: $A_1(A_2A_3 \ldots A_n)$, $(A_1A_2)(A_3 \ldots A_n)$,...,$(A_1A_2A_3 \ldots A_{n-1})A_n$. We take the least-costly out of all of those
  - We get:

$$R(1,n) = \min_{1 \le i \le n-1} \Big(d_0 d_i d_n + R(1,i) + R(i+1,n) \Big)$$

    Base case: $R(i,i) = 0$ for any i
- Fairly simple to see this recursion yields a non-efficient algorithm (Not so simple to see — this recursion's runtime is proportional to the Catalan number of n: $\frac{1}{n+1}\left(\begin{array}{c} 2n \\ n \end{array}\right) \in \Omega(3^n)$)
- Cannot afford this runtime...
- But luckily, all recurrences live in a small domain: Contiguous sequences of multiplying $A_i \times A_{i+1} \times \ldots \times A_j$. ($\binom{n}{2}$ options)

Matrix-chain multiplication: 2^{nd} Solution — Dynamic Programming

- Step 1: Define $M[i,j]$ $(1 \le i \le j)$: the minimum number of scalar multiplications needed to compute product $A_i \times A_{i+1} \times \ldots \times A_j$ $(i \le j)$
- Step 2: The recurrence to fill in the entries of the array:

$$M[i,j] = \begin{cases} 0, & \text{if } i = j \\ \min_{i \le k < j} \Big(M[i,k] + M[k+1,j] + d_{i-1} d_k d_j \Big), & \text{if } i < j \end{cases}$$

- for example,

$$M[1,4] = \min \left\{ \begin{array}{l} M[1,1] + M[2,4] + d_0 d_1 d_4 \\ M[1,2] + M[3,4] + d_0 d_2 d_4 \\ M[1,3] + M[4,4] + d_0 d_3 d_4 \end{array} \right\}$$

- Note: When we fill $M[i,j]$ we want all the required $M[i,k]$ and $M[k,j]$ cells to be filled.
- So what is the bottom-up fashion here?
  - We know how to fill $M[i,i]$ for every i.
  - Based on this, we will fill $M[i,i+1]$ for every $i \le n-1$.
  - Based on those, we will fill $M[i,i+2]$ for every $i \le n-2$.
  - And so on and so forth
  - I.e. bottom-up — using the gap between the i and j.

Matrix-chain multiplication: 2^{nd} Solution — Dynamic Programming

- Pseudocode (to obtain the optimal cost & optimal order):

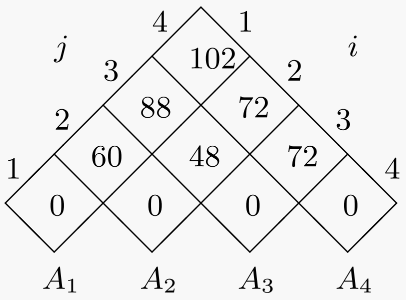
```
procedure QuickestMultiplication(d_0, d_1, ..., d_n)
for (i ← 1 to n) do
    M[i, i] ← 0
    S[i, i] ← ⊥
for (gap ← 1 to n − 1) do
    for (i ← 1 to n − gap) do
        M[i, i + gap] ← ∞
        for (k ← 0 to gap − 1) do
            cut_at_k ← M[i, i + k] + M[i + k + 1, i + gap] + d_{i−1} × d_{i+k} × d_{i+gap}
            if (cut_at_k < M[i, i + gap]) then
                M[i, i + gap] ← cut_at_k
                S[i, i + gap] ← i + k        ** Memorizing the least expensive break-point
return M[1, n]
```

- To obtain the actual ordering we call `Print-Opt-Order`$(M, 1, n)$:

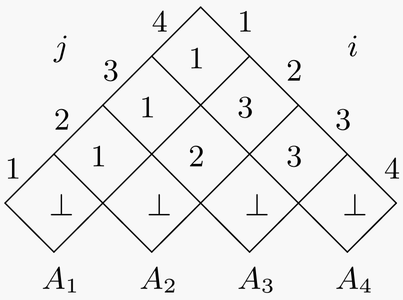
```
procedure Print-Opt-Order(S, i, j)
if (i = j) then
    Print (''A''_i)
else
    Print ''(''
        Print-Opt-Order (S, i, S[i, j])        ** The indentation is just for the ease of reading.
    Print '')×(''
        Print-Opt-Order (S, S[i, j] + 1, j)
    Print '')''
```

- Trace the example $n = 4$ and $(d_0, d_1, d_2, d_3, d_4) = (5, 2, 6, 4, 3)$:

M-matrix



S-matrix



- Runtime:
  - The innermost `for` loopbody takes constant time ...
  - We iterated over gap, i and k, each takes no more that n options.
  - Hence, runtime of $O(n^3)$.
  - Moreover, for $gap \in [\frac{n}{3}, \frac{2n}{3}]$, i iterates on at least $\frac{n}{3}$ options and k iterates over at least $\frac{n}{3}$ options, so runtime is at least $\Omega(\frac{n}{3} \cdot \frac{n}{3} \frac{n}{3}) = \Omega(n^3)$.
  - Hence, running time $\in \Theta(n^3)$.

rod cutting

length	1	2	3	4	5	6	7	8
price	1	5	8	9	10	17	17	20

$c_i = \max(V_k + c_{i-k})$

$$c_2 = \begin{cases} V_1 + c_1 = 2 \\ 5 \end{cases}$$

$$c_3 = \begin{cases} V_1 + c_2 = 1+5 = 6 \\ V_1 + V_1 + V_1 = 1+1+1 = 3 \\ V_3 = 8 \end{cases}$$

$$c_4 = \max \begin{cases} V_1 + c_3 = 1+8 = 9 \\ V_2 + c_2 = 5+5 = 10 \\ V_3 + V_1 = 8+1 = 9 \\ V_4 = 9 \end{cases} \quad 10$$

7 1 5 3 6 4

0 1 4 4 5

-2 1 -3 7 -2 2 1 -5 4

-2 1 -2 7 5 7 8 3 7

1 8 6 2 5 4 8 3 7