Dynamic Programming

- Coin Changing
- Knapsack Problem
- Matrix Chain Multiplication

Coin Changing

Coin Changing

Goal. Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.

Ex: 34¢.

Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Ex: $2.89.

Coin-Changing: Greedy Algorithm

Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

```
Sort coins denominations by value: c_1 < c_2 < … < c_n.

     coins selected
S ← φ
while (x ≠ 0) do
   let k be largest integer such that c_k ≤ x
   if (k = 0) then
      return "no solution found"
   x ← x - c_k
   S ← S ∪ {k}
return S
```

Q. Is cashier's algorithm optimal?

Counterexample. Coins: 1, 7, 9 and $x = 14$

Greedy algorithm: 9, 1, 1, 1, 1, 1

Optimal: 7, 7

Def. OPT(x) = min number of coins needed to change x using coins $c_1 < c_2 < \ldots < c_n$

- Imagine we change x step by step and in each step we use a coin of n given coins. Indeed, we check all n possible cases in each step. Therefore

$$OPT(x) = \begin{cases} 1 & \text{if } \exists i : \mathrm{x} = \mathrm{c}_i \\ \min\{1 + OPT(\mathrm{x} - \mathrm{c}_i)\} & \text{otherwise} \end{cases}$$

Dynamic Programming

Filling Strategy. Fill up a 1-to-x array C bottom-up

```
Input c_1 < c_2 < … < c_n and x

for k = 1 to x do
    C(k) = +infinity
for i = 1 to n do
    C(c_i) = 1
for k = 1 to x do
    for i = 1 to n do
       if (x-c_i>0) and (C[k] > C[x-c_i]) then
          C[k] = C[x-c_i]+1
return C(x)
```

Running Time

Running time. $O(nx)$

Q. Is the running time polynomial based on the input size?

Assume coins are 1, 5 and $x = 10^{20}$. The input is just 3 numbers and the running time is $O(10^{20})$. Is it really polynomial? If you think it is polynomial set $x = 10^{100}$ and wait million years to get the optimal solution only when your input is 3 numbers.

Input Size. n+1 numbers. If we assume each number has l bits, the input size in $(n+1)l$.

The worst-case running time. If $x = 2^l$, the running time is $O(n\,2^l)$ Which is not polynomial based on the input size $(n+1)l$.

The running time is Pseudo-polynomial, i.e., it is polynomial based the value of the input variables.

Knapsack Problem

Knapsack Problem

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: { 3, 4 } has value 40.

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Greedy: repeatedly add item with maximum ratio v_i / w_i.
Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

Dynamic Programming: False Start

Def. OPT(i) = max profit subset of items 1, …, i.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 }

- Case 2: OPT selects item i.
  - accepting item i does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before i, we don't even know if we have enough room for i

Conclusion. Need more sub-problems!

Dynamic Programming: Adding a New Variable

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w
- Case 2: OPT selects item i.
  - new weight limit = w – w_i
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } \; i = 0 \\ OPT(i-1, w) & \text{if } \; w_i > w \\ \max\{\, OPT(i-1, w), \;\; v_i + OPT(i-1, w-w_i) \,\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

Knapsack. Fill up an n-by-W array.

```
Input: n, W, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

Knapsack Algorithm

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Problem: Running Time

Running time. $\Theta(n\ W)$.

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

Knapsack approximation algorithm. There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

Matrix Chain Multiplication

Matrix Multiplication

- Let A be a matrix of dimension pxq and B be a matrix of dimension qxr.
- Then, if we multiply matrices A and B, we obtain a resulting matrix C= AB whose dimension is pxr,
- We can obtain each entry in C using q scalar multiplications. In total, pqr scalar multiplications.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix} \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \end{pmatrix} = \begin{pmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{pmatrix}$$

Matrix Multiplication

- Associative property. We know $((A_1A_2)A_3) = (A_1(A_2A_3))$.
- Then, any way to write down the parentheses gives the same result.
  Example. $(((A_1A_2)A_3)A_4) = ((A_1A_2)(A_3A_4)) = (A_1((A_2A_3)A_4)) =$
  $((A_1(A_2A_3))A_4) = (A_1(A_2(A_3A_4)))$
- The number of scalar multiplications may be different due to different computation sequence.
- Let the dimensions of A_1, A_2, A_3 be: 1x100, 100x1, 1x100, respectively.
  # scalar multiplications to get $((A_1A_2)A_3)$ = 1x100x1+1x1x100
  # scalar multiplications to get $(A_1(A_2A_3))$ = 100x1x100+1x100x100

Matrix Chain Multiplication

Problem. Given a chain of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, where A_i has dimensions $p_{i-1} \times p_i$, fully parenthesize the product $A_1 \cdot A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

$$\begin{array}{ccccccccccc} A_1 & \cdot & A_2 & \cdots & A_i & \cdot & A_{i+1} & \cdots & A_n \\ p_0 \times p_1 & & p_1 \times p_2 & & p_{i-1} \times p_i & & p_i \times p_{i+1} & & p_{n-1} \times p_n \end{array}$$

- Exhaustively checking all possible parenthesizations is not efficient!
- It can be shown that the number of parenthesizations grows as $\Omega(4^n/n^{3/2})$

The Structure of an Optimal Parenthesization

Notation. $A_{i\ldots j} = A_i\, A_{i+1} \cdots A_j,\ i \le j$

Observation.

- Suppose that an optimal parenthesization of $A_{i\ldots j}$ splits the product between A_k and A_{k+1}, where $i \le k < j$

$$\begin{aligned} A_{i\ldots j} &= A_i\, A_{i+1} \cdots A_j \\ &= A_i\, A_{i+1} \cdots A_k\, A_{k+1} \cdots A_j \\ &= A_{i\ldots k}\, A_{k+1\ldots j} \end{aligned}$$

- The parenthesization of $A_{i\ldots k}$ and $A_{k+1\ldots j}$ must be optimal.

An optimal solution to an instance of the matrix-chain multiplication contains within it optimal solutions to subproblems,

A Recursive Solution

Subproblem. determine the minimum cost of parenthesizing

$A_{i\ldots j} = A_i\, A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$

The optimal solution. Let $M[i, j]$ = the minimum number of multiplications needed to compute $A_{i\ldots j}$

- full problem ($A_{1..n}$): $M[1, n]$
- $i = j$: $A_{i\ldots i} = A_i \Rightarrow M[i, i] = 0$ for $i = 1, 2, \ldots, n$

A Recursive Solution

- Consider the subproblem of parenthesizing

  $A_{i\ldots j} = A_i\, A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$

  $= A_{i\ldots k}\, A_{k+1\ldots j}$ for $i \le k < j$

- Assume that the optimal parenthesization splits the product $A_i\, A_{i+1} \cdots A_j$ at k ($i \le k < j$). Then,

$M[i, j] =$ $M[i,k]+$ $M[k+1, j]+$ $p_{i-1}p_kp_j$

min # of multiplications to compute $A_{i\ldots k}$

min # of multiplications to compute $A_{k+1\ldots j}$

# of multiplications to compute $A_{i\ldots k}A_{k\ldots j}$

We do not know the value of k

- There are $j - i$ possible values for k: $k = i, i+1, \ldots, j-1$
- Minimizing the cost of parenthesizing the product $A_i\, A_{i+1} \cdots A_j$ becomes: $M[i, j] = \min_{i \le k < j} \{M[i, k] + M[k+1, j] + p_{i-1}p_kp_j\}$ if $i < j$

Computing the Optimal Costs

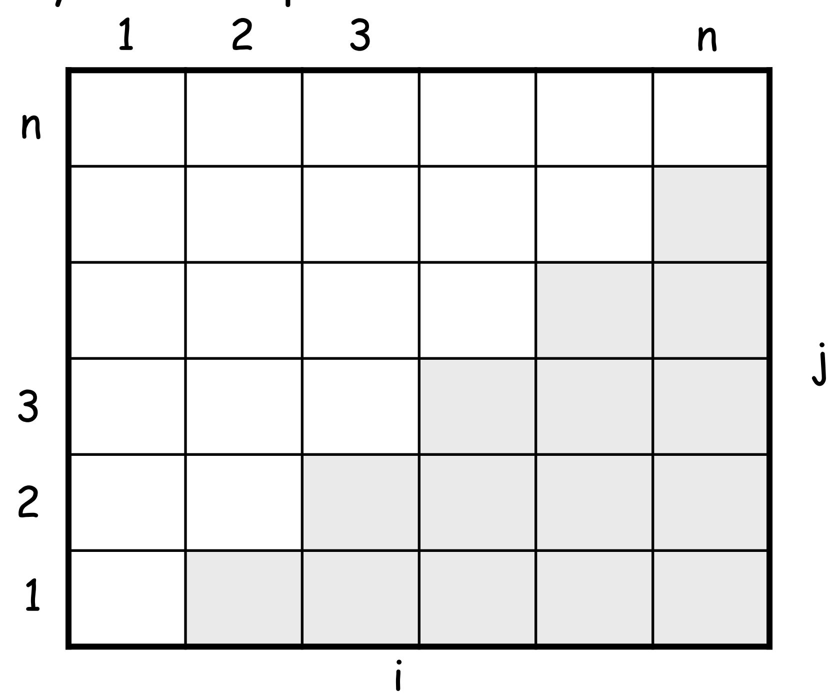
$$M(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min\{M[i,k] + M[k+1,j] + p_{i-1}p_j p_k\} & \text{if } i < j \end{cases}$$

Computing the optimal solution recursively takes exponential time!

How many subproblems?

$\Rightarrow \Theta(n^2)$

- Parenthesize $A_{i \ldots j}$ for $1 \le i \le j \le n$
- One problem for each choice of i and j



Computing the Optimal Costs

$$M(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min\{M[i,k] + M[k+1,j] + p_{i-1}p_jp_k\} & \text{if } i < j \end{cases}$$

How do we fill in the tables M[1..n, 1..n]?

- Determine which entries of the table are used in computing M[i, j]

$$A_{i\ldots j} = A_{i\ldots k}\, A_{k+1\ldots j}$$

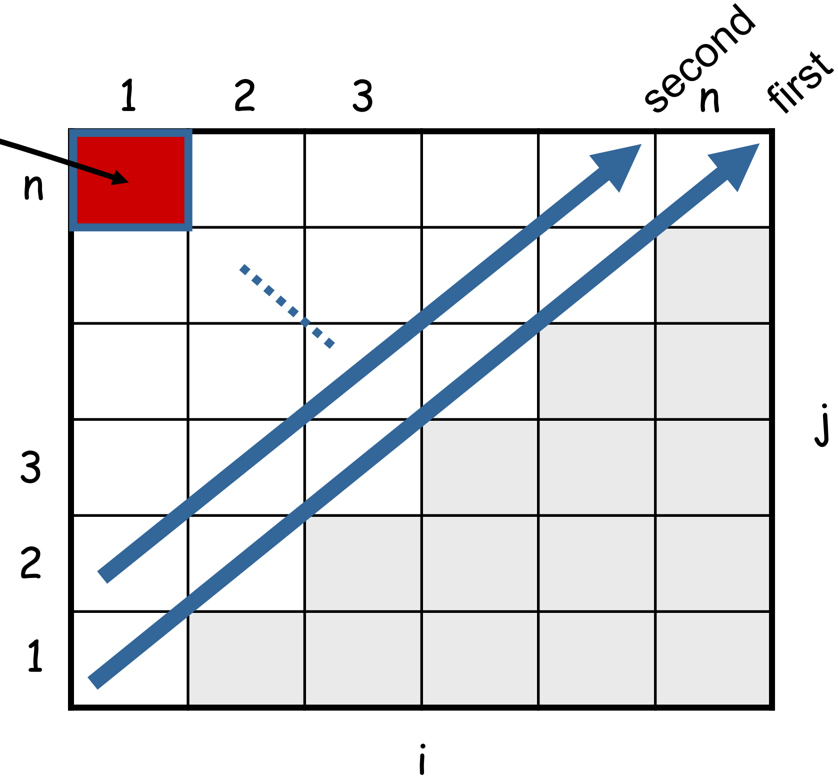
- Subproblems' size is one less than the original size
- Idea: fill in M such that it corresponds to solving problems of increasing length

i

Computing the Optimal Costs

$$M(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min\{M[i,k] + M[k+1,j] + p_{i-1}p_jp_k\} & \text{if i < j} \end{cases}$$

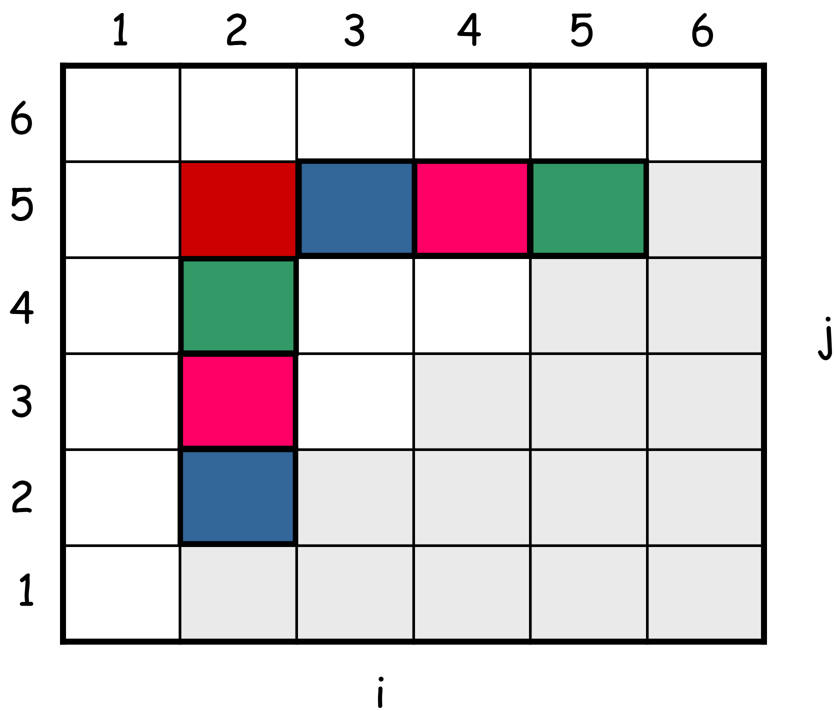
M[1, n] gives the optimal solution to the problem



Compute rows from bottom to top and from right to left

Computing the Optimal Costs

$$M[2, 5] = \min \begin{cases} M[2,2] + M[3,5] + p_1 p_2 p_5 & k = 2 \\ M[2,3] + M[4,5] + p_1 p_3 p_5 & k = 3 \\ M[2,4] + M[5,5] + p_1 p_4 p_5 & k = 4 \end{cases}$$



- Values $m[i, j]$ depend only on values that have been previously computed

Filling Strategy. Fill up a n-to-n array M bottom-up

```
Input  Matrices A1, A2, … , An

for j = 1 to n do
  for i = j to 1 do
     M[i,j] = +infinity
for j = 1 to n do
    M[j,j] = 0
for j = 1 to n do
  for i = j to 1 do
    for k = j-1 to i do
       M[i,j] = min(M[i,j], M[i,k]+M[k+1,j]+pi-1pjpk)
return M[1,n]
```

Running time. $O(n^3)$

References

References

- Section 6.4 of the text book "algorithm design" by Jon Kleinberg and Eva Tardos
- Section 15.2 of the text book "introduction to algorithms" by CLRS, 3rd edition.
- The original slides were prepared by Kevin Wayne. The slides are distributed by Pearson Addison-Wesley.
- The matrix-chain-multiplication part is from the slides prepared by George Bebbis.