Algorithms (2IL15) – Lecture 3

DYNAMIC PROGRAMMING
Optimization problems

- for each instance there are (possibly) multiple valid solutions
- goal is to find an optimal solution

- minimization problem:
  associate cost to every solution, find min-cost solution
- maximization problem:
  associate profit to every solution, find max-profit solution
Techniques for optimization

Optimization problems typically involve making choices.

**Backtracking:** just try all solutions
- Can be applied to almost all problems, but gives very slow algorithms
- Try all options for first choice,
  for each option, recursively make other choices

**Greedy algorithms:** construct solution iteratively, always make choice that seems best
- Can be applied to few problems, but gives fast algorithms
- Only try option that seems best for first choice (greedy choice), recursively make other choices

**Dynamic programming**
- In between: not as fast as greedy, but works for more problems
Algorithms for optimization: how to improve on backtracking

for greedy algorithms

1. try to **discover** structure of optimal solutions: what **properties** do optimal solutions have?
   - what are the choices that need to be made?
   - do we have **optimal substructure**?
     - optimal solution = first choice + optimal solution for subproblem
   - do we have **greedy-choice** property for the first choice?

what if we do not have a good greedy choice?

then we can often use **dynamic programming**
Dynamic programming: examples

- maximum weight independent set (non-adjacent vertices) in path graph

- optimal matrix-chain multiplication: today

- more examples in book and next week
Matrix-chain multiplication

Input: $n$ matrices $A_1, A_2, \ldots, A_n$

Required output: $A_1 \cdot A_2 \cdot \ldots \cdot A_n$

Cost: $a \cdot b \cdot c$ scalar multiplications $= \Theta(a \cdot b \cdot c)$ time
Matrix-chain multiplication

Multiplying three matrices:

\[
A \cdot B \cdot C
\]

\[
\begin{align*}
2 \times 10 & & 10 \times 50 & & 50 \times 20 \\
\end{align*}
\]

Does it matter in which order we multiply?
Note for answer: \((A \cdot B) \cdot C = A \cdot (B \cdot C)\) matrix multiplication is associative:

What about the cost?
- \((A \cdot B) \cdot C\) costs \((2 \cdot 10 \cdot 50) + (2 \cdot 50 \cdot 20) = 3,000\) scalar multiplications
- \(A \cdot (B \cdot C)\) costs \((10 \cdot 50 \cdot 20) + (2 \cdot 10 \cdot 20) = 10,400\) scalar multiplications
Matrix-chain multiplication: the optimization problem

Input: \( n \) matrices \( A_1, A_2, \ldots, A_n \), where \( A_i \) is \( p_{i-1} \times p_i \) matrix

In fact: we only need \( p_0, \ldots, p_n \) as input

Required output: an order to multiply the matrices minimizing the cost

in other words: the best way to place brackets

So we have

valid solution = complete parenthesization: \( ((A_1 \cdot A_2) \cdot (A_3 \cdot (A_4 \cdot (A_5 \cdot A_6)))) \)

cost = total number of scalar multiplications needed to multiply

according to the given parenthesization
Let’s study the structure of an optimal solution

- what are the choices that need to be made?
- what are the subproblems that remain? do we have optimal substructure?

\[
\begin{align*}
\left( A_1 \cdot (A_2 \cdot A_3) \right) \cdot \left( (A_4 \cdot A_5) \cdot (A_6 \cdot A_7) \right)
\end{align*}
\]

**choice: final multiplication**

**optimal solution**

**optimal way to multiply** \( A_1, \ldots, A_3 \)

**optimal way to multiply** \( A_4, \ldots, A_7 \)

**optimal substructure:** optimal solution is composed of optimal solutions to subproblems of same type as original problem

necessary for dynamic programming (and for greedy)
Let’s study the structure of an optimal solution

- what are the choices that need to be made?
- what are the subproblems that remain? do we have optimal substructure?

choice: final multiplication

Do we have a good greedy choice? 

no ...

then what to do?

try all possible options for position of final multiplication
Approach:

- try all possible options for final multiplication
- for each option recursively solve subproblems

Need to specify exactly what the subproblems are

**Initial problem:** compute \( A_1 \cdot \ldots \cdot A_n \) in cheapest way

\[
(A_1 \cdot \ldots \cdot A_k) \cdot (A_{k+1} \cdot \ldots \cdot A_n)
\]

**Subproblem 1:** compute \( A_1 \cdot \ldots \cdot A_k \) in cheapest way

\[
(A_1 \cdot \ldots \cdot A_i) \cdot (A_{i+1} \cdot \ldots \cdot A_k)
\]

**Subproblem 2:** compute \( A_{i+1} \cdot \ldots \cdot A_k \) in cheapest way

**General form:** given \( i, j \) with \( 1 \leq i < j \leq n \), compute \( A_i \cdot \ldots \cdot A_j \) in cheapest way
Now let’s prove that we indeed have optimal substructure

**Subproblem:** given $1 \leq i < j \leq n$, compute $A_i \cdot \ldots \cdot A_j$ in cheapest way

Define $m[i,j] = \text{value of optimal solution to subproblem}$

= minimum cost to compute $A_i \cdot \ldots \cdot A_j$

**Lemma:** $m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \} & \text{if } i < j \end{cases}$

but we don’t know the best way to decompose into subproblems
Lemma: \[ m[i,j] = \begin{cases} 
0 & \text{if } i = j \\
\min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & \text{if } i < j 
\end{cases} \]

Proof:

\( i=j \): trivial

\( i<j \): Consider optimal solution \( \text{OPT} \) for computing \( A_i \cdot \ldots \cdot A_j \) and let \( k^* \) be the index such that \( \text{OPT} \) has the form \( (A_i \cdot \ldots \cdot A_{k^*}) \cdot (A_{k^*+1} \cdot \ldots \cdot A_j) \). Then

\[ m[i,j] = (\text{cost in } \text{OPT for } A_i \cdot \ldots \cdot A_{k^*}) + (\text{cost in } \text{OPT for } A_{k^*+1} \cdot \ldots \cdot A_n) + p_{i-1} p_{k^*} p_j \]

\[ \geq m[i,k^*] + m[k^*+1,j] + p_{i-1} p_{k^*} p_j \]

\[ \geq \min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} \]

On the other hand, for any \( i \leq k < j \), there is a solution of cost \( m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \) so \( m[i,j] \leq \min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} \).

“cut-and-paste” argument: replace solution that \( \text{OPT} \) uses for subproblem by optimal solution to subproblem, without making overall solution worse.
Lemma: $m[i,j] = \begin{cases} 
0 & \text{if } i = j \\
\min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \} & \text{if } i < j 
\end{cases}$

$RMC(p,i,j)$

\begin{algorithm}
    \text{initial call: } RMC(p[0..n],1,n)

    1. \text{if } i = j
    2. \text{then}
    3. \hspace{1em} \text{cost} \leftarrow 0
    4. \text{else}
    5. \hspace{1em} \text{cost} \leftarrow \infty
    6. \hspace{1em} \text{for } k \leftarrow i \text{ to } j-1
    7. \hspace{2em} \text{do } \text{cost} \leftarrow \min(\text{cost}, RMC(p,i,k) + RMC(p,k+1,j) + p_{i-1}p_kp_j)
    8. \hspace{1em} \text{return } \text{cost}
\end{algorithm}
Correctness:

by induction and lemma on optimal substructure

Analysis of running time:

Define \( T(m) \) = worst-case running time of \( RMC(i,j) \), where \( m = j - i +1 \).

Then:

\[
T(m) = \begin{cases} 
\Theta(1) & \text{if } m = 1 \\
\Theta(1) + \sum_{1 \leq s < m} \{ T(s) + T(m-s) + \Theta(1) \} & \text{if } m >1
\end{cases}
\]

Claim: \( T(m) = \Omega (2^m) \)

Proof: by induction (exercise)

Algorithm is faster than completely brute-force, but still very slow …
Why is the algorithm so slow?

recursive calls

(1,1) (2,n)  (1,2) (3,n)  (1,3) (4,n)  …  (1,n-1) (n,n)

(2,2) (3,n)  (2,3) (4,n)  …  (2,n-1) (n,n)

(3,3) (4,n)  …

overlapping subproblems, so same subproblem is solved many times

optimal substructure + overlapping subproblems: then you can often use dynamic programming
How many different subproblems are there?

subproblem: given $i,j$ with $1 \leq i \leq j \leq n$, compute $A_i \cdot \ldots \cdot A_j$ in cheapest way

number of subproblems = $\Theta(n^2)$
**Lemma:** \( m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & \text{if } i < j \end{cases} \)

**Idea:** fill in the values \( m[i,j] \) in a table (that is, an array) \( m[1..n,1..n] \).

Fill in the table in an order so that the entries needed to compute a new entry are already available.

value of optimal solution to original problem

or or …
Lemma: \( m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \} & \text{if } i < j \end{cases} \)

MatrixChainDP (\( p[0..n] \))

\( \text{algorithm uses two-dimensional table (array) } m[1..n,1..n] \)

to store values of solutions to subproblems

1. \( \text{for } i \leftarrow 1 \text{ to } n \)
2. \( \text{do } m[i,i] \leftarrow 0 \)
3. \( \text{for } i \leftarrow n-1 \text{ downto } 1 \)
4. \( \text{do for } j \leftarrow i + 1 \text{ to } n \)
5. \( \text{do } m[i,j] \leftarrow \min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \} \)
6. \( \text{return } m[1,n] \)
Analysis of running time

\[ \sum_{1 \leq i \leq n-1} \sum_{i+1 \leq j \leq n} \Theta(j-i) = \sum_{1 \leq i \leq n-1} \sum_{1 \leq j \leq n} \Theta(j) = \sum_{1 \leq i \leq n-1} \Theta((n-i)^2) = \sum_{1 \leq i \leq n-1} \Theta(i^2) = \Theta(n^3) \]

MatrixChainDP ( \( p[0..n] \) )

1. for \( i \leftarrow 1 \) to \( n \)
2. do \( m[i,i] \leftarrow 0 \)
3. for \( i \leftarrow n-1 \) downto \( 1 \)
4. do for \( j \leftarrow i+1 \) to \( n \)
5. do \( m[i,j] \leftarrow \min\{ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \} \)
6. return \( m[1,n] \)

for each of the \( \Theta(n^2) \) cells we spend \( \Theta(n) \) time
Designing dynamic-programming algorithms

1. Investigate structure of optimal solution: do we have optimal substructure, no greedy choice, and overlapping subproblems?

2. Give recursive formula for the value of an optimal solution
   i. define subproblem in terms of a few parameters
   given $i,j$ with $1 \leq i < j \leq n$, compute $A_i \cdot \ldots \cdot A_j$ in cheapest way
   ii. define variable $m[\cdot,\cdot] = \text{value of optimal solution for subproblem}$
   $m[i,j] = \text{minimum cost to compute } A_i \cdot \ldots \cdot A_j$
   iii. give recursive formula for $m[\cdot,\cdot]$
   
   $$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & \text{if } i < j \end{cases}$$

3. Algorithm: fill in table for $m[\cdot,\cdot]$ in suitable order

4. Extend algorithm so that it computes an actual solution that is optimal, and not just the value of an optimal solution
4. Extend algorithm so that it computes an actual solution that is optimal, and not just the value of an optimal solution

   i. introduce extra table that remembers choices leading to optimal solution

   \[ m[i,j] = \text{minimum cost to compute } A_i \cdot \ldots \cdot A_j \]

   \[ s[i,j] = \text{index } k \text{ such that there is an optimal solution of the form } (A_i \cdot \ldots \cdot A_k) \cdot (A_{k+1} \cdot \ldots \cdot A_j) \]

   ii. modify algorithm such that it also fills in the new table

   iii. write another algorithm that computes an optimal solution by walking “backwards” through the new table
MatrixChainDP( p[0..n] )

1. for i ← 1 to n
2. do m[i,i] ← 0
3. for i ← n-1 downto 1
4. do for j ← i +1 to n
5. do m[i,j] ← \min \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \}
   \quad \text{for } i \leq k < j
6. return m[1,n]

\( m[i,j] \leftarrow \infty \)

for k ← i to j-1

\text{do begin}
\quad cost \leftarrow m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
\quad \text{if } cost < m[i,j] \text{ then } m[i,j] \leftarrow cost
\quad s[i,j] \leftarrow k
\text{end}
s[i,j] = index k such that there is an optimal solution of the form

\((A_i \cdot \ldots \cdot A_k) \cdot (A_{k+1} \cdot \ldots \cdot A_j)\)

to compute \(A_i \cdot \ldots \cdot A_j\)

```
PrintParentheses (i, j)
1. if i = j
2. then print “A_i”
3. else begin
4. print “(“
5. PrintParentheses (i, s[i,j])
6. PrintParentheses (s[i,j]+1, j)
7. print “)”
8. end
```
Alternative to filling in table in suitable order: **memoization**
recursive algorithm, but remember which subproblems have been solved already

**initialize array** $m[i,j]$ (global variable): $m[i,j] = \infty$ for all $1 \leq i,j \leq n$

**$RMC(i,j)$**

1. if $i = j$
2. then
3. return 0
4. else
5. $cost \leftarrow \infty$
6. for $k \leftarrow i$ to $j-1$
7. do $cost \leftarrow \min(m, RMC(i,k) + RMC(k+1,j) + p_{i-1} p_k p_j)$
8. return $cost$

**if** $m[i,j] < \infty$
**then** return $m[i,j]$
**else** lines 4 – 7

$m[i,j] \leftarrow cost$
Summary dynamic programming

1. Investigate structure of optimal solution: do we have optimal substructure, no greedy choice, and overlapping subproblems?

2. Give recursive formula for the value of an optimal solution
   i. define subproblem in terms of a few parameters
   ii. define variable $m[..] = \text{value of optimal solution for subproblem}$
   iii. give recursive formula for $m[..]$

3. Algorithm: fill in table for $m[..]$ in suitable order

4. Extend algorithm so that it computes an actual solution that is optimal, and not just the value of an optimal solution

next week more examples of dynamic programming