Matrix Chain Multiplication

The input to the following algorithm is $p = (p_0, \ldots, p_n)$, where $p_{i-1} \times p_i$ is the dimension of $A_i$.

- $s[i, j]$ is the best place to split the computation of $A_{i..j}$ to $A_{i..k}A_{k+1..j}$.

**Matrix-Chain-Order($p$)**

1. $n \leftarrow \text{length}[p] - 1$
2. for $i \leftarrow 1$ to $n$ do
3. \hspace{1em} $m[i, j] \leftarrow 0$
4. for $l \leftarrow 2$ to $n$ do
5. \hspace{2em} for $i \leftarrow 1$ to $n - l + 1$ do
6. \hspace{3em} $j \leftarrow i + l - 1$
7. \hspace{3em} $m[i, j] \leftarrow \infty$
8. \hspace{2em} for $k \leftarrow i$ to $j - 1$ do
9. \hspace{3em} $q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$
10. \hspace{3em} if $q < m[i, j]$
11. \hspace{4em} then $m[i, j] \leftarrow q$
12. \hspace{3em} $s[i, j] \leftarrow k$
13. return $m$ and $s$

Running time: $O(n^3)$

- Key point: the same information ($m[i, j]$) gets reused over and over.
Computing an optimal solution

\textbf{Matrix-Chain-Order} computes the best place to split and the optimal number of scalar multiplications.

- From $s[i, j]$, it’s easy to compute how to multiply

\textbf{M-Chain-Multiply}(A, s, i, j)

\begin{verbatim}
1   if \( j > i \) 2       then \( X \leftarrow \text{M-Chain-Multiply}(A, s, i, s[i, j]) \)
3                   \( Y \leftarrow \text{M-Chain-Multiply}(A, s, s[i, j] + 1, j) \)
4   return \text{Matrix-Multiply}(X, Y) 
5   else return \( A_i \)
\end{verbatim}

Get the right answer by calling \textbf{M-Chain-Multiply}(A, s, 1,
Longest Common Subsequence

Given two sequences, we want to find there longest common subsequence. This is a problem that comes up, for example, in gene sequencing (if we want to compare to genomes).

Formally, if \( Z = (z_1, \ldots, z_k) \) is a subsequence of \( X = (x_1, \ldots, x_m) \) if there exist \( i_1, \ldots, i_k \) such that \( i_1 < \ldots < i_k \) and \( z_j = x_{i_j} \).

Example: The longest common subsequence of \( (A, A, B, C, A, A, D, A) \) and \( (A, C, B, C, A, B, D, C, A) \) is \( (A, B, C, A, D, A) \).

- There can be at most 3 A’s in the lcs, so this is the best we can do.

The brute force approach to finding LCS of \( X \) and \( Y \) is to consider all subsequences of \( X \) and see which ones are subsequences of \( Y \).

- The number of subsequences of \( X \) is exponential in \( \text{length}(X) \).

We can do better using dynamic programming.
Characterizing an LCS

Given a sequence $X = (x_1, \ldots, x_m)$, if $i \leq m$, let $X_i = (x_1, \ldots, x_i)$.

**Theorem:** Suppose that $Z = (z_1, \ldots, z_k)$ is an lcs for $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k-1}$ is an lcs for $X_{m-1}$ and $Y_{n-1}$.

2. If $x_m \neq y_n$ and $z_k \neq x_m$, then $Z$ is an lcs for $X_{m-1}$ and $Y_n$.

3. If $x_m \neq y_n$ and $z_k \neq y_n$, then $Z$ is an lcs for $X$ and $Y_{n-1}$.

Therefore, an lcs for $X$ and $Y$ contains within it an lcs for two smaller sequences.

- We can find $\text{LCS}(X, Y)$ by first finding $\text{LCS}(X_i, Y_j)$ for all the prefixes of $X$ and $Y$. 
Solving LCS Recursively

Let $c[i, j]$ the length of an lcs of $X_i$ and $Y_j$.

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0, x_i = y_j \\ \max(c[i - 1, j], c[i, j - 1]) & \text{if } i, j > 0, x \neq y_j \end{cases}$$

LCS-LENGTH($X, Y$)

1. $n \leftarrow \text{length}[X]$
2. $m \leftarrow \text{length}[Y]$
3. for $i \leftarrow 1$ to $m$ do
4. \hspace{1em} $c[i, 0] \leftarrow 0$
5. for $j \leftarrow 0$ to $n$ do
6. \hspace{1em} $c[0, j] \leftarrow 0$
7. for $i \leftarrow 1$ to $m$ do
8. \hspace{1em} for $j \leftarrow 1$ to $n$ do
9. \hspace{2em} if $x_i = y_j$
10. \hspace{3em} then $c[i, j] \leftarrow c[i - 1, j - 1] + 1$
11. \hspace{2em} else $c[i, j] \leftarrow \max(c[i - 1, j], c[i, j - 1])$
12. return $c$

Running time (and space): $O(nm)$
Printing out an LCS

\textsc{Print-LCS}(c, X, i, j)

1 \textbf{if} \ i = 0 \text{ or } j = 0
2 \quad \textbf{then return}
3 \quad \textbf{if} \ c[i - 1, j] = c[i, j]
4 \quad \textbf{then} \ \textsc{Print-LCS}(c, X, i - 1, j)
5 \quad \textbf{else if} \ c[i, j - 1] = c[i, j]
6 \quad \textbf{then} \ \textsc{Print-LCS}(c, X, i, j - 1)
7 \quad \textbf{else} \ \textsc{Print-LCS}(c, X, i - 1, j - 1)
8 \quad \text{print } x_i
Greedy Algorithms

One approach to an optimization problem: make the choice that currently looks best.

- Sometimes this greedy approach is a bad idea
  - you can get caught in a trap
- Other times it works remarkably well.

Kruskal’s algorithm for MST can be viewed as a greedy algorithm:

- Choose the edge of least weight that buys you something

So can Prim’s algorithm:

- Choose the edge of least weight that extends the current tree and buys you something.

And so can Dijkstra’s algorithm:

- Choose the node not yet chosen which is closest to the source.
Activity Selection

Suppose that we have a set $S = \{1, \ldots, n\}$ of proposed activities that need to use the same resource

- only one can be active at a time
  
  - example: scheduling classes in a lecture hall
- Activity $i$ has a start time $s_i$ and a finish time $f_i$.

Problem: choose the maximum set of mutually compatible activities

- Don’t want activities whose start-finish times overlap

Basic idea: keep choosing an activity as long as it’s compatible with the ones you’ve already chosen.

- The actual algorithm suggests a particular way to choose.
Order the activities by increasing finish time:

\[ f_1 \leq f_2 \leq \ldots \leq f_n \]

- This pre-processing step takes time \( O(n \log n) \)

Assume the algorithm gets as input the sequence \( s \) of start times and the sequence \( f \) of finish times (in sorted order):

**Greedy-Activity-Selector** \((s, f)\)

1. \( n \leftarrow \text{length}[s] \)
2. \( A \leftarrow \{1\} \quad [A \text{ consists of selected activities}] \)
3. \( j \leftarrow 1 \quad [j \text{ is the last activity put into } A] \)
4. **for** \( j \leftarrow 2 \) **to** \( n \) **do**
5.  \( \text{if } s_i \geq f_j \quad [\text{if it's safe to add } i \text{ to } A \ldots] \)
6.  \( \text{then } A \leftarrow A \cup i \)
7.  \( j \leftarrow i \)
8. **return** \( A \)

Clearly this gives a set of compatible activities.

It's also efficient:

- After preprocessing, run in time \( \Theta(n) \).

But why is it correct?
**Theorem:** GReedy-ACTIVITY-SELECTOR chooses a maximum set of mutually compatible activities.

**Proof:** By strong induction on $n$, the number of activities in $S$.

Base case: clearly OK if $S = 1$.

Inductive step: First show that there is a maximum set that includes activity 1 (the one with earliest finish time).

Let $A$ be a maximum set and let $k$ be the activity in $A$ with earliest finish time.

- If $k = 1$, we’re done.
- If not, let $B = A - \{k\} \cup \{1\}$. The activities in $B$ must be mutually compatible
  - activity 1 can’t overlap with anything, since its finish time is earlier than $k$’s
- Thus, $B$ is a maximum set that includes 1.
If $A$ is a maximum set of mutually compatible activities in $S = \{1, \ldots, n\}$ and $1 \in A$, then $A - \{1\}$ is a maximum set of mutually compatible activities in $S' = \{i \in S : s_i \geq f_1\}$.

- $S'$ consists of activities that start after 1 ends.

Now by induction, the algorithm produces a maximum set on $S'$.

- But the action of algorithm on $S'$ is exactly the same as the action of the algorithm on $S$ after choosing 1.
Greedy vs. Dynamic Programming

A greedy algorithm works only if making the greedy choice gives an optimal solution:

- That works in some cases, but not always.
- The hard part is often showing that it works

Example:

- The 0-1 knapsack problem: there are $n$ items
  - Item $i$ has value $v_i$ and weight $w_i$.

  You can put at most $W$ pounds into a knapsack. Which items do you take?
  - For each item, you either take it or leave it (0-1)

- The fractional knapsack problem: same setup, but now you can take part of an item.
  - This means you have more flexibility

Key point:

- There’s a greedy algorithm for the fractional knapsack problem, but not for the 0-1 knapsack problem
For the fractional knapsack problem:

- First sort the items by value/pound \(v_i/w_i\)
- Pick the most valuable items that you can fit in, then the next one, etc., until there’s no more room.
- Then put in as much of the last item as you can to get to weight \(W\).
  - This is OK since you can take fractions of an item.

This approach doesn’t work for the 0-1 knapsack problem:

- Suppose there are three items and the knapsack can hold 50 pounds:
  - Item 1 weighs 10 lb. and is worth $60
  - Item 2 weighs 20 lb. and is worth $100
  - Item 3 weighs 30 lb. and is worth $120
- Item 1 is the most valuable, but the optimal solution is \(\{2, 3\}\).

You can use dynamic programming to solve the 0-1 knapsack problem.
Huffman Codes

Suppose you have a large file, where only 6 different characters appear

- Not all characters appear equally often
- How do we represent the characters so as to get greatest compression?
  - Compression is critical in transmitting data over a modem
  - There are *lots* of coding algorithms

Assume each character is represented as a binary string. Example:

\[
\begin{align*}
a &= 000000 \\
b &= 000001 \\
z &= 011010 \\
, &= 011011
\end{align*}
\]

Is this a good encoding?

- This is a fixed-length code: all characters encoded by a 6-bit code word
- It’s a better idea to use a variable-length code
- Greater frequency \(\Rightarrow\) shorter code word
  - Modern coding algorithms (based on Ziv-Lempel) adaptively choose length of code word
Prefix Code

If one code is a prefix of another, then decoding is harder

- if \( e \) is 0 and \( a \) is 01, when you see 0, is it an \( e \) or the beginning of an \( a \).

It is best to assume a *prefix code*

- no codeword is the prefix of another codeword.

Decoding is simple with a prefix code:

- Keep running along string until you have a complete codeword, and continue
  - Note: this is a greedy decoding algorithm

- E.g., suppose \( e = 0, a = 10, b = 110 \)
  - then 00110100 = *eebae*