Fundamental Algorithms

Brute force, Greedy, Dynamic Programming:
Dynamic Programming Technique

- Primarily for optimization problems

- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as $j$, $k$, $l$, $m$, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).
Matrix Chain-Products

Let's start from a mathematic problem

- Matrix Multiplication.
  - \( C = A \times B \)
  - \( A \) is \( d \times e \) and \( B \) is \( e \times f \)
  - \( A \times B \) takes \( d \times e \times f \) times of basic operations

\[
C[i, j] = \sum_{k=0}^{e-1} A[i, k] \times B[k, j]
\]
Matrix Chain-Products

- Compute $A = A_0 * A_1 * \ldots * A_{n-1}$
- $A_i$ is $d_i \times d_{i+1}$
- Problem: We want to find a way to compute the result with the minimal number of operations
Matrix Chain-Products

- How to put parentheses on matrix?

- Example:
  - B is $3 \times 100$
  - C is $100 \times 5$
  - D is $5 \times 5$
  - $(B*C)*D$ takes $1500 + 75 = 1575$ ops
  - $B*(C*D)$ takes $1500 + 2500 = 4000$ ops
  - The order of computation matters!
  - We want to figure out the way with the minimal cost
Brute-force

- An enumeration approach

Matrix Chain-Product Alg.:
- Try all possible ways to parenthesize \( A = A_0 \cdot A_1 \cdot \ldots \cdot A_{n-1} \)
- Calculate number of ops for each one
- Pick the one that is best

Running time:
- The number of paranethesizations is equal to the number of binary trees with \( n \) nodes
- This is **exponential**!
- It is called the Catalan number, and it is almost \( 4^n \).
- This is a terrible algorithm!
Greedy

- Choose the local optimal iteratively
- Repeatedly select the product that uses the fewest operations.

Example:
- A is $10 \times 5$
- B is $5 \times 10$
- C is $10 \times 5$
- D is $5 \times 10$
- $A \times B$ or $B \times C$ or $C \times D \rightarrow B \times C$
- $A \times ((B \times C) \times D)$ takes $500 + 250 + 250 = 1000$ ops
Another example

- Another example
  - A is $101 \times 11$
  - B is $11 \times 9$
  - C is $9 \times 100$
  - D is $100 \times 99$

- The greedy approach gives $A \ast ((B \ast C) \ast D))$, which takes $109989 + 9900 + 108900 = 228789$ ops

- However, $(A \ast B) \ast (C \ast D)$ takes $9999 + 89991 + 89100 = 189090$ ops

- This is a counter example that the greedy approach does not give us an optimal solution
Dynamic Programming

- Simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner

Two key observations:

- The problem can be split into sub-problems

- The optimal solution can be defined in terms of optimal sub-problems
Dynamic Programming

- Find the best parenthesization of $A_i * A_{i+1} * \ldots * A_j$.
- Let $N_{i,j}$ denote the number of operations done by this subproblem.
- The optimal solution for the whole problem is $N_{0,n-1}$.
- There has to be a final multiplication (root of the expression tree) for the optimal solution.
- Say, the final multiply is at index $i$: $(A_0 * \ldots * A_i) * (A_{i+1} * \ldots * A_{n-1})$. 
Dynamic Programming

- Then the optimal solution $N_{0,n-1}$ is the sum of two optimal subproblems, $N_{0,i}$ and $N_{i+1,n-1}$ plus the time for the last multiply.

- If the global optimum did not have these optimal subproblems, we could define an even better “optimal” solution.

- We can compute $N_{i,j}$ by considering each $k$. 
A Characterizing Equation

- Let us consider all possible places for that final multiply:
  - Recall that $A_i$ is a $d_i \times d_{i+1}$ dimensional matrix.
  - So, a characterizing equation for $N_{i,j}$ is the following:

$$N_{i,j} = \min_{i \leq k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_k d_{k+1} d_{j+1} \}$$

- Note that sub-problems overlap and hence we cannot divide the problem into completely independent sub-problems (divide and conquer)
Bottom-up computation

- \( N(i,i) = 0 \)
- \( N(i,i+1) = N(i,i) + N(i+1,i+1) + d_i d_{i+1} d_{i+2} \)
- \( N(i,i+2) = \min \{ \)
  \( N(i,i) + N(i+1,i+2) + d_i d_{i+1} d_{i+2} \)
  \( N(i,i+1) + N(i+2,i+2) + d_i d_{i+2} d_{i+2} \)
\)
- \( N(i,i+3) \ldots \)
- Until you get \( N(i,j) \)
A Dynamic Programming Algorithm Visualization

- The bottom-up construction fills in the N array by diagonals.
- \( N_{i,j} \) gets values from previous entries in i-th row and j-th column.
- Filling in each entry in the N table takes \( O(n) \) time.
- Total run time: \( O(n^3) \)
- Getting actual parenthesization can be done by remembering “k” for each N entry.

\[
N_{i,j} = \min_{i \leq k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}
\]
A Dynamic Programming Algorithm

- Since subproblems overlap, we don’t use recursion.
- Instead, we construct optimal subproblems “bottom-up.”
- \( N_{i,i} \)’s are easy, so start with them
- Then do length 2, 3, … subproblems, and so on.
- The running time is \( O(n^3) \)

Algorithm \( \text{matrixChain}(S) \):

**Input:** sequence \( S \) of \( n \) matrices to be multiplied

**Output:** number of operations in an optimal paranethization of \( S \)

for \( i \leftarrow 1 \) to \( n-1 \) do

\[ N_{i,i} \leftarrow 0 \]

for \( b \leftarrow 1 \) to \( n-1 \) do

for \( i \leftarrow 0 \) to \( n-b-1 \) do

\[ j \leftarrow i+b \]

\[ N_{i,j} \leftarrow +\text{infinity} \]

for \( k \leftarrow i \) to \( j-1 \) do

\[ N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\} \]
Similarity between strings

- A common text processing problem:
  - Two strands of DNA
  - Two versions of source code for the same program
  - diff (a built-in program for comparing text files)
Subsequences

- A *subsequence* of a character string $x_0x_1x_2\ldots x_{n-1}$ is a string of the form $x_{i_1}x_{i_2}\ldots x_{i_k}$, where $i_j < i_{j+1}$.

- Not necessary contiguous but taken in order

- Not the same as substring!

- Example String: ABCDEFGHIJK
  - Subsequence: ACEGIJK
  - Subsequence: DFGHK
  - Not subsequence: DAGH
The Longest Common Subsequence (LCS) Problem

- Given two strings X and Y, the longest common subsequence (LCS) problem is to find a longest subsequence common to both X and Y.
- Has applications to DNA similarity testing (alphabet is \{A,C,G,T\}).
- Example: ABCDEFG and XZACKDFWGH
  - have ACDFG as a longest common subsequence.
A Poor Approach to the LCS Problem

- A Brute-force solution:
  - Enumerate all subsequences of X
  - Test which ones are also subsequences of Y
  - Pick the longest one.

- Analysis:
  - If X is of length $n$, then it has $2^n$ subsequences
  - If Y is of length $m$, the time complexity is $O(2^{nm})$
  - This is an exponential-time algorithm!
A Dynamic-Programming Approach to the LCS Problem

- Define $L[i,j]$ to be the length of the longest common subsequence of $X[0..i]$ and $Y[0..j]$.

- Allow for -1 as an index, so $L[-1,k] = 0$ and $L[k,-1] = 0$, to indicate that the null part of $X$ or $Y$ has no match with the other.

- Then we can define $L[i,j]$ in the general case as follows:
  1. If $x_i = y_j$, then $L[i,j] = L[i-1,j-1] + 1$ (we can add this match)
  2. If $x_i \neq y_j$, then $L[i,j] = \max\{L[i-1,j], L[i,j-1]\}$ (we have no match here)
A Dynamic-Programming Approach to the LCS Problem

Case 1:

\[ L[8,10] = 5 \]

Case 2:

\[ L[9,9] = 6 \]

\[ L[8,10] = 5 \]
An LCS Algorithm

Algorithm LCS(X,Y):
Input: Strings X and Y with n and m elements, respectively
Output: For i = 0,…,n-1, j = 0,…,m-1, the length L[i, j] of a longest string that is a subsequence of both the string X[0..i] = x0x1x2…xi and the string Y [0.. j] = y0y1y2…yj

for i =0 to n-1 do
    L[i,-1] = 0
for j =0 to m-1 do
    L[-1,j] = 0
for i =0 to n-1 do
    for j =0 to m-1 do
        if x_i = y_j then
            L[i, j] = L[i-1, j-1] + 1
        else
            L[i, j] = max{L[i-1, j] , L[i, j-1]}
return array L
## Visualizing the LCS Algorithm

<table>
<thead>
<tr>
<th></th>
<th>-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>122</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>122</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>111</td>
<td>122</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>111</td>
<td>122</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>111</td>
<td>122</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>111</td>
<td>122</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>111</td>
<td>122</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>111</td>
<td>122</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>111</td>
<td>122</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
<td>222</td>
</tr>
</tbody>
</table>

$X = \text{GTTCCCTAATA}$

$Y = \text{CGATAATTGAGA}$
Analysis of LCS Algorithm

- We have two nested loops
  - The outer one iterates \( n \) times
  - The inner one iterates \( m \) times
  - A constant amount of work is done inside each iteration of the inner loop
  - Thus, the total running time is \( O(nm) \)

- Answer is contained in \( L[n,m] \) (and the subsequence can be recovered from the L table).
Exercise

Given two strings, output the LCS

- Example:
  - Inputs: “Fang Yu” and “Shannon Yu”
  - Output: “an Yu”
if $x_i = y_j$ then
  $L[i, j] = L[i-1, j-1] + x_i$;
else
  $L[i, j] = (L[i-1, j].size() <= L[i, j-1].size())?L[i, j-1]:L[i-1, j]$;

return $L[n-1,m-1]$ ;
HW9 (Due on Dec. 21)

Find the most similar keyword!

- Implement the LCS algorithm for keywords
- Add each keyword into an array/linked list
- Given a string s, output the keyword k, such that k’s value and s have the longest common sequence among all the added keywords.
Operations

Given a sequence of operations in a txt file, parse the txt file and execute each operation accordingly

<table>
<thead>
<tr>
<th>operations</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(Keyword k)</td>
<td>Insert a keyword k to an array</td>
</tr>
<tr>
<td>find(String s)</td>
<td>Find and output the most similar keyword by using the LCS algorithm</td>
</tr>
</tbody>
</table>
An input file

Similar to HW9,

1. You need to read the sequence of operations from a txt file
2. The format is firm
3. Raise an exception if the input does not match the format

add Fang 3
add Yu 5
add NCCU 2
add UCSB 1
add Management 4
add Information 5
find NTU
find Manager

NTU: [NCCU, 2]
Manager: [Management, 4]