Dynamic Programming

Ref: Chapter 15 of text.

Computing Fibonacci Numbers:
Recall that $F_0 = 0$, $F_1 = 1$ and
$$F_n = F_{n-1} + F_{n-2} \quad \text{for all } n \geq 2.$$ 

An “easy” recursive algorithm:

Fibo (n) // n >= 0.
if (n <= 1)
    then return n
else return Fibo(n-1) + Fibo(n-2)

Recurrence for running time:

$$T(n) = T(n-1) + T(n-2) \quad \text{for } n \geq 2$$
with $T(0) = c_1$ and $T(1) = c_2$ for some constants $c_1$ and $c_2$.

Exercise: Use induction on $n$ to show that for all $n \geq 3$, $T(n) \geq \Phi^{n-2}$, where $\Phi = (1 + \sqrt{5})/2 > 1$.

Observation: The running time of the recursive algorithm is exponential in $n$.

Reason for inefficiency: The same value gets computed many times.

Remedy:

- Avoid recomputation of values by saving them in a table and use them when they are needed.
- A key idea of dynamic programming.

Iterative algorithm: (Time: $O(n)$)

Note: A[0..n] stores the computed values.

1. A[0] = 0; A[1] = 1
2. for i = 2 to n do
Matrix-Chain Multiplication:

Simple facts:

- For the matrix product $A \times B$ to be defined, the number of columns of $A$ must be equal to the number of rows of $B$.
- Suppose $A$ is a $p \times q$ matrix and $B$ is a $q \times r$ matrix. The common method of computing the product $A \times B$ uses $pqr$ multiplications.
- Matrix multiplication is associative; that is, for any three matrices $A$, $B$ and $C$,
  \[
  A \times (B \times C) = (A \times B) \times C
  \]
  as long as all the matrix products are defined.

Note: From now on, matrix product $A \times B$ will be denoted by $AB$.

A Motivating Example: Suppose $A$, $B$ and $C$ are matrices of the following sizes:

- $A$: $r \times 2$
- $B$: $2 \times r$
- $C$: $r \times 1$

Method I: Compute the product $ABC$ as $(AB)C$.

- Computing $AB$ uses $2r^2$ multiplications.
- Computing the product of $AB$ with $C$ uses $r^2$ multiplications.
- Total number of multiplications $= 3r^2$.

Method II: Compute the product as $A(BC)$.

- Computing $BC$ uses $2r$ multiplications.
- Computing the product of $A$ with $BC$ uses $2r$ multiplications.
- Total number of multiplications $= 4r$.

Conclusion: Method II is better.

Note: Each method of evaluation corresponds to fully parenthesizing the product.
Example: With four matrices $A$, $B$, $C$ and $D$, some ways of fully parenthesizing the product $ABCD$ are:

$((AB)C)D$, $(AB)(CD)$,
$(A(BC'))D$, $A(B(CD))$

Matrix-Chain Multiplication Problem:

Given: A chain $\langle A_1 A_2 \cdots A_n \rangle$ of matrices where $A_i$ is a $p_{i-1} \times p_i$ matrix ($1 \leq i \leq n$).

Required: Fully parenthesize the product $A_1 A_2 \cdots A_n$ so that the total number of scalar multiplications used is minimized.

Initial focus: Compute the minimum number of multiplications.

Known Result: The number of ways of fully parenthesizing the above product is at least

$C(2n,n)/(n+1) = \Omega(4^n/n^{3/2})$.

So, exhaustive search is not viable.

Approach: Let $M = A_1 A_2 \cdots A_n$. In any optimal solution, the last step evaluates the product $M_1 M_2$ where

$M_1 = A_1 A_2 \cdots A_k$ and $M_2 = A_{k+1} A_{k+2} \cdots A_n$

for some $k$ in the range $1$ to $n-1$.

Observation: Regardless of the value of $k$, for the evaluation of $M$ to be optimal, the evaluations of $M_1$ and $M_2$ must themselves be done optimally.

What does the observation suggest?

- The original problem gives rise to subproblems of smaller sizes.
- To obtain an optimal solution to the original problem, the subproblems must also be solved optimally.
- Save solutions to subproblems to avoid recomputation.
What do we store for each subproblem?

- Each subproblem consists of a subchain $C_{ij} = A_i A_{i+1} \cdots A_j$, where $1 \leq i \leq j \leq n$. (So, the number of subproblems = $O(n^2)$, which is a polynomial.)
- An $n \times n$ array $M$ used to store solutions to subproblems; that is, $M[i, j]$ stores the minimum number of multiplications needed to evaluate the subchain $C_{ij}$.
- For each $i$, $1 \leq i \leq n$, $M[i, i] = 0$.
- If all entries of $M$ are available, the solution to the problem is given by $M[1, n]$.

Computing entries of $M$:

Say, we want to compute $M[i, j]$ for some $i < j$.

- Consider all possible values $k$ where the subchain $A_i A_{i+1} \cdots A_j$ can be split. Thus, the value of $k$ varies from $i$ to $j - 1$.

- If the split is at $k$, the following costs are involved:
  1. $M[i, k]$ for evaluating the subchain $A_i \cdots A_k$.
  2. $M[k + 1, j]$ for evaluating the subchain $A_k+1 \cdots A_j$.
  3. Cost $p_{i-1}p_kp_j$ to multiply the two matrices resulting from (a) and (b).

So, the value of $M[i, j]$ is the minimum of

$$M[i, k] + M[k + 1, j] + p_{i-1}p_kp_j \quad (1)$$

over the range of $k$ values, that is, $i \leq k < j$.

Organizing the Computation:

- Initialization: $M[i, i] = 0$, $1 \leq i \leq n$.
- To compute $M[i, j]$ using Expression (1) above, we need to make sure that all the required entries of $M$ have already been computed.
So, compute the entries of $M$ in *increasing order* of the subscript difference $j - i$ as follows.

(i) First, compute all entries where the subscript difference is zero; that is, compute $M[i, i], 1 \leq i \leq n$. (This is trivial since $M[i, i] = 0, 1 \leq i \leq n$.)

(ii) Next, compute all entries where the subscript difference is one; that is, compute $M[i, i + 1], 1 \leq i \leq n - 1$.

(iii) Then compute all entries where the subscript difference is two; that is, compute $M[i, i + 2], 1 \leq i \leq n - 2, \ldots$, and so on up to subscript difference $n - 1$.

Now, when we need to compute $M[i, j]$ with $j - i = t$, we will have the values of $M[i, k]$ and $M[k + 1, j]$ where the subscript difference is less than $t$.

**Pseudocode:** See Handout 13.1.

**Numerical example:** To be presented in class.

**Running time:** (Step numbers are from Handout 13.1.)

- **Step 1:** $O(n)$.
- **Step 2:**
  - Steps 2.1.1 and 2.1.2 take $O(n)$ time.
  - The loop in Step 2.1 executes $O(n)$ times and each iteration takes $O(n)$ time. So, the time for Step 2.1 is $O(n^2)$.
  - The loop in Step 2 executes $O(n)$ times and each iteration takes $O(n^2)$ time. So, the time for Step 2 is $O(n^3)$.
- **Step 3:** $O(1)$ time.
- **So,** the overall running time is $O(n^3)$.

**Space:** $O(n^2)$ due to the $n \times n$ matrix $M$.

**Exercise:** Modify the pseudocode in Handout 13.1 so that a fully parenthesized expression is also printed out in addition to the minimum number of multiplications. The running time should remain $O(n^3)$. 
Hint: Make each entry of $M[i,j]$ a record containing two fields: one field is the cost as before and the other field is a value of index $k$ where an optimal split of the subchain $C_{ij} = A_i A_{i+1} \cdots A_j$ occurs.

Best known algorithm: $O(n \log n)$ due to T. Hu and M. Shing (SIAM J. Computing, 1982).

Maximum Subsequence Sum Problem:

- Considered in Lecture 4.
- Given a sequence $S$ of numbers $\langle x_1, x_2, \ldots, x_n \rangle$, a subsequence is a contiguous part of $S$, namely $\langle x_i, x_{i+1}, \ldots, x_k \rangle$ for some $i \leq k$.
- A single value is a subsequence of length 1. (We will ignore the empty subsequence.)

Problem Statement:

Given: A sequence $S$ of $n$ numbers $\langle x_1, x_2, \ldots, x_n \rangle$, some of which may be negative.

Required: The maximum sum over all subsequences of $S$.

Approach:

- Subproblem: For each $i$, find the best subsequence sum that ends at $x_i$, $1 \leq i \leq n$. (So, there are $n$ subproblems.)
- Solution to subproblems stored in array $B[1..n]$.

Computation Steps:

- To compute $B[i+1]$ for $1 \leq i \leq n-1$:
  
  $B[i+1] = B[i] + x_{i+1}$ if $B[i] > 0$
  
  $= x_{i+1}$ otherwise.

- Solution is the largest value in $B$.

Pseudocode: Handout 13.1

Running time and space: $O(n)$.

Exercise: Modify the pseudocode in Handout 13.1 so as to print out a subsequence with the maximum sum. The running time should remain $O(n)$.
**Subset Sum Problem:**

**Recall:** A multi-set may contain duplicate values.

**Problem statement:**

Given: A multi-set $S = \{a_1, a_2, \ldots, a_n\}$ of positive integers and a positive integer $B$.

Question: Is there a subset $S'$ of $S$ such that the sum of the elements in $S'$ is equal to $B$?

**Note:** The Subset Sum problem is NP-complete.

**Dynamic Programming Approach:**

- Each subproblem consists of the multi-set $\{a_1, \ldots, a_i\}$ and integer $j$, where $1 \leq i \leq n$ and $0 \leq j \leq B$.
- Solutions to subproblems are stored in a two-dimensional Boolean array $T$ with $n$ rows (numbered 1 to $n$) and $B + 1$ columns (numbered 0 through $B$). The significance of $T[i, j]$ is as follows:

  - $T[i, j] = true$ if there is a subset of $\{a_1, \ldots, a_i\}$ with sum $= j$
  - $T[i, j] = false$ otherwise.

- After computing all entries of $T$, the Boolean value $T[n, B]$ tells us whether there is a solution to the problem.

**Computation Steps:**

- All entries in the first row of $T$ can be computed easily:

  - $T[1, j] = true$ if $j = 0$ or $j = a_1$
  - $T[1, j] = false$ otherwise.

- To compute the entries in row $i$ for $2 \leq i \leq n$:

  - $T[i, j] = true$ if $T[i - 1, j] = true$ or $T[i - 1, j - a_i] = true$
  - $T[i, j] = false$ otherwise.

**Pseudocode:** Handout 13.1.
Running time:

- There are $O(nB)$ entries in array $T$.
- The value of each entry can be computed in $O(1)$ time.
- So, the overall running time is $O(nB)$.
- The size of input is $O(n \log a + \log B)$, where $a$ is the largest integer in $S$.
- So, the running time is not polynomial in the size of the input.

Space:

- Space is also $O(nB)$ due to the array $T$.
- Can be reduced to $O(B)$ since we need to keep only two rows of $T$. 