CS3230 Chapter 5 - Dynamic Programming

Based on lectures by Chang Ee-Chien
Notes taken by Andrew Tan
AY18/19 Semester 1

1 Dynamic Programming

Like dividing-and-conquering, dynamic programming is a general method of algorithmic design. It is usually applied to optimization problems, especially in cases where divide-and-conquer algorithms are inefficient due to repetitions in computing sub-problems. Dynamic programming modifies the divide-and-conquer algorithm to a *bottom-up* algorithm to avoid computing these sub-problems repeatedly.

We explore an organized method for applying dynamic programming, which consists of a four-step sequence:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

These steps help us in developing a dynamic-programming algorithm, and they go a long way in our study in solving optimization problems.

In this chapter we'll look at a few examples, namely calculating combinations, matrix chain multiplication, string matching, and memoization.

2 Combinations

Given integers n and k, we would like to compute $\binom{n}{k}$ using the property below:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Using recursion, we can use the following top-down algorithm:

```
choose(n, k):
  if n == k or k == 0 then return 1
  if k < 0 or n < k then return 0
  return choose(n − 1, k − 1) + choose(n − 1, k)
```

Let $T(n, k)$ be the number of arithmetic additions required. Then

$$T(n, k) = T(n-1, k-1) + T(n-1, k) + 1$$

The running time of the algorithm itself is roughly $O(n^k)$.

Now let's explore a bottom up approach using dynamic programming:

```
choose_dynamic(n, k):
   A = array[n + 1][n + 1]
   for i ← 0 to n:
      A[i, 0] ← 1
      A[i, i] ← 1
   for i ← 1 to n:
      for j ← 1 to min(i, k) + 1:
         A[i, j] ← A[i − 1][j] + A[i − 1][j − 1]
   return A[n, k]
```

This essentially constructs Pascal's triangle, which we can then easily obtain $\binom{n}{k}$.
The running time of this algorithm is in $\Theta(nk)$.

3 Matrix Chain Multiplication

Given a sequence $\{A_1, A_2, \ldots, A_n\}$ of n matrices to be multiplied, we wish to compute the product $A_1 A_2 \ldots A_n$.

We can evaluate the product using the standard algorithm for multiplying pairs of matrices, though we do need to parenthesize the product to specify how the matrices are multiplied together.

How we parenthesize a chain of matrices can have a significant impact on the cost of evaluating the product, and as such the **matrix-chain multiplication problem** is concerned with finding the parenthesization of a matrix chain such that the number of scalar multiplications are minimized.

Using a top down approach to exhaustively check all possible parenthesizations, we obtain the following recurrence relation:

$$P(n) = \begin{cases} 1, & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k), & \text{if } n \geq 2 \end{cases}$$

The solution to the recurrence is $\Omega(2^n)$, and thus the number of solutions is exponential in n.

3.1 Applying dynamic programming

We will walk through the steps to solve this problem, and demonstrate how we apply each step to the problem.

3.1.1 The structure of an optimal parenthesization

Suppose that to optimally parenthesize $A_i A_{i+1} \ldots A_j$ for $i < j$, we split the product between A_k and A_{k+1} for some $i, j, k, i \leq k < j$. Then the way we parenthesize the "prefix" subchain $A_i A_{i+1} \ldots A_k$ within this optimal parenthesization of $A_i A_{i+1} \ldots A_j$ must also be an optimal parenthesization of $A_i A_{i+1} \ldots A_k$.

Thus, with our optimal substructure defined, we can construct an optimal solution to the problem from optimal solutions to subproblems. We must also ensure that when we search for the correct place to split the product, we have considered all possible places, so that we are certain of having examined the optimal one.

3.1.2 A recursive solution

Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the product of the matrices $A_i \ldots A_j$. We can define $m[i, j]$ recursively as follows. If $i = j$ the problem is trivial, and thus $m[i, i] = 0$ for $i = 1, 2, \ldots, n$. To compute $m[i, j]$ when $i < j$, we take advantage of the optimal structure as described above. Thus, we obtain $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$. Where $p_{i-1} p_k p_j$ denote the number of scalar multiplications it takes to compute the matrix product $(A_i \ldots A_k)(A_{k+1} A_j)$. We do not know the actual value for k, and thus we will need to check

all possible values for k.

Thus, we obtain our recursive definition for the minimum cost of parenthesizing the product $A_iA_{i+1}\dots A_j$ as follows:

$$m[i,j] = \begin{cases} 0, & \text{if } i = j \\ \min\limits_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}, & \text{if } i < j \end{cases}$$

3.1.3 Computing the optimal costs

We could easily write a recursive algorithm based on the above recurrence. However, this algorithm takes exponential time. Now observe that we have relatively few distinct subproblems: one subproblem for each choice of i and j satisfying $1 \le i \le j \le n$, or $\binom{n}{2} + n = \Omega(n^2)$. A recursive algorithm may encounter each subproblem many times, and hence we can apply dynamic programming here as a more effective approach.

We compute the optimal cost by using a tabular, bottom-up approach. Our procedure assume that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \dots, n$. We input a sequence $p = \langle p_0, p_1, \dots, p_n \rangle$, where $p.length = n + 1$. We also use an auxilliary table $m[1 \dots n-1, 2 \dots n]$ for storing the $m[i,j]$ costs and another auxilliary table $s[1 \dots n-1, 2 \dots n]$ that records which index of k achieved the optimal cost in computing $m[i,j]$. We use the table s to construct an optimal solution.

Thus, this algorithm fills in the table m in a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length.

```
Matrix-Chain-Order(p):
n = p.length − 1
let m[1 . . . n, 1 . . . n] and s[1 . . . n − 1, 2 . . . n] be new tables
for i = 1 to n:
   m[i, i] = 0
for l = 2 to n: // l is the chain length
   for i = 1 to n − l + 1:
      j = i + l − 1
      m[i, j] = ∞
      for k = i to j − 1:
         q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
         if q < m[i, j]:
            m[i, j] = q
            s[i, j] = k
return m, s
```

We can see that the running time of $Matrix - Chain - Order$ yields a running time of $O(n^3)$ for the algorithm, and requires $\Theta(n^2)$ space to store the m and s tables.

3.1.4 Constructing an optimal solution

$Matrix - Chain - Order$ does not directly show how to multiply the matrices, but the table $s[1 \dots n-1, 2 \dots n]$ gives us the information we need to do so. Each entry $s[i,j]$ records a value of k such that an optimal parenthesization of $A_iA_{i+1}\dots A_j$ splits the product between A_k and A_{k+1}. Thus, we know that the final matrix multiplication in computing $A_1 \dots A_n$ optimally is $(A_1 \dots A_{s[1,n]})(A_{s[1,n]+1} \dots A_n)$. We can also determine the earlier matrix multiplications recursively, since $s[1, s[1,n]]$ determines the last matrix multiplication when computing $A_1 \dots A_{s[1,n]}$ and $s[s[1,n]+1, n]$ determines the last matrix multiplication when computing $A_{s[1,n]+1} \dots A_n$.

Now, we can design a procedure that prints the optimal parenthesization of $\langle A_1, A_2, \dots, A_n \rangle$.

```
Print-Optimal-Parens(s, i, j):
if i == j:
   print "A"
else
```

```
print "("
Print-Optimal-Parens(s, i, s[i, j])
Print-Optimal-Parens(s, s[i, j] + 1, j)
print ")"
```

4 Longest common subsequence

Given a sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$, $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a **subsequence** of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$. For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$.

Thus, given two sequences X and Y, we wish to find a sequence Z such that it is the longest common subsequence (LCS) of X and Y.

We can use a brute-force approach to solve this problem by simply checking for all subsequences of X if it is also a subsequence of Y. Since X has 2^m subsequences, this approach requires exponential time, making it impractical for long sequences.

4.1 Applying dynamic programming

4.1.1 Characterizing a longest common subsequence

For a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, let us define the i^{th} **prefix** of X, for $i = 0, 1, \ldots, m$, as $X_i = \langle x_1, x_2, \ldots, x_i \rangle$.

Let us now discuss the optimal substructure of an LCS. Let $X = \langle x_1, x_2, \ldots, x_\rangle$ and $Y = \langle y_1, y_2, \ldots y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y. Then the following theorem defines the optimal substructure of an LCS: **Optimal substructure of an LCS**

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

The way that this theorem characterizes longest common subsequences tells us that an LCS of two sequences contains within it an LCS of prefixes of the two sequences. Thus, the LCS problem has an optimal-substructure property.

4.1.2 A recursive solution

The theorem implies that we should examine either one or two subproblems when finding an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots y_n \rangle$. We can readily see the overlapping-subproblems property in the LCS problem. To find an LCS of X and Y, we may need to find the LCSs of X and Y_{n-1} and of X_{m-1} and Y. Each of these subproblems has the subsubproblem of finding an LCS of X_{m-1} and Y_{n-1}.

Now we can establish a recursive solution to the LCS problem. Let us define $c[i, j]$ be the length of an LCS of the sequences X_i and Y_j. The optimal substructure of the LCS problem yields the recursive formula

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

4.1.3 Computing the length of an LCS

Based on the recurrence relation above, we can easily write an exponential-time recursive algorithm to compute the length of an LCS of two sequences. Since the LCS has only $\Theta(mn)$ distinct subproblems, however, we can use dynamic programming to compute the solutions bottom up.

Our method will take in two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots y_n \rangle$. It then stores the $c[i, j]$ values in a table $c[0 \ldots m, 0 \ldots n]$, and it computes the entries in *row-major* order (that is, it fills in the first row of c from left to right, then the second row, and so on). The procedure also maintains the table $b[1 \ldots m, 1 \ldots n]$ to help us construct an optimal solution. $b[i, j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i, j]$. The method then returns b and c, and $c[m, n]$ contains the length of an LCS of X and Y.

```
LCS-Length(X, Y):
m = X.length
n = Y.length
let b[1...m, 1...n] and c[1...m, 1...n] be new tables
for i = 1 to m:
   c[i, 0] = 0
for j = 0 to n:
   c[0, j] = 0
for i = 1 to m:
   for j = 1 to n:
      if x_i == y_i:
         c[i, j] = c[i − 1, j − 1] + 1
         b[i, j] = " ↖ "
      else if c[i − 1, j] ≥ c[i, j − 1]:
         c[i, j] = c[i − 1, j]
         b[i, j] = " ↑ "
      else:
         c[i, j] = c[i, j − 1]
         b[i, j] = " ← "
return c, b
```

The running time of the procedure is $\Theta(mn)$, since each table entry takes $\Theta(1)$ time to compute.

4.1.4 Constructing an LCS

The b table returns by the method enables us to quickly construct an LCS of X and Y. We simply begin at $b[m, n]$ and trace through the table by following the arrows. Whenever we encounter a " ↖ " in entry $b[i, j]$, it implied that $x_i = y_i$ is an element of the LCS that the method has found. With this method, we encounter the elements of this LCS in reverse order.

The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The intial call is Print-LCS($b, X, X.length, Y.length$).

```
Print-LCS(b, X, i, j):
if i == 0 or j == 0:
   return
if b[i, j] == " ↖ ":
   Print-LCS(b, X, i − 1, j − 1)
   print x_i
else if b[i, j] == " ↑ ":
   Print-LCS(b, X, i − 1, j)
else:
   Print-LCS(b, X, i, j − 1)
```

5 Memoization

Oftentimes, it is difficult to design bottom-up algorithms, as we need to know how to effectively compute and populate the respective tables in the correct order. Furthermore, some problems do not require all subproblems to be calculated beforehand.

Thus, we can combine our divide and conquer top-down approach and our tabulating approach with memoization.

As the problems that we've encountered have many overlapping subproblems, we can simply modify our recursive algorithms to keep track of the values that have been computed, and refer to our table once we've encountered it again.