Dynamic Programming 1: Introduction

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong
This is the beginning of several lectures on the topic of **dynamic programming**. This technique aims to avoid repetitive computation in solving a problem recursively, and often allows us to reduce the running time from an exponential function to a polynomial function.
A Recurrence Computation Problem

**Input:** An array $A$ that contains $n$ integers.

**Output:** Compute the value of $F(1, n)$, where for any $i, j \in [1, n]$

$$F(i, j) =
\begin{cases}
0 & \text{if } i > j \\
\left(\sum_{k=i}^{j} A[k]\right) + \min_{k=i}^{j} \left\{ F(i, k - 1) + F(k + 1, j) \right\} & \text{otherwise}
\end{cases}$$
Example: Suppose that $A = (40, 15, 35, 10)$
We have:

- $F(1, 0) = 0$
- $F(1, 1) = 40, F(2, 2) = 15, F(3, 3) = 35, F(4, 4) = 10$
- $F(1, 2) = 70, F(2, 3) = 65, F(3, 4) = 55$
- $F(1, 3) = 155, F(2, 4) = 85$
- $F(1, 4) = 180$
Naive Recursion

The recurrence

\[ F(i, j) = \begin{cases} 
0 & \text{if } i > j \\
\left( \sum_{k=i}^{j} A[k] \right) + \min_{k=i} \left\{ F(i, k - 1) + F(k + 1, j) \right\} & \text{otherwise}
\end{cases} \]

leads to a straightforward recursive algorithm:

\textbf{algorithm } F(i, j) \\
1. \textbf{if } i > j \textbf{ return } 0 \\
2. \text{\textit{common}} = \sum_{k=i}^{j} A[k] \\
3. \text{\textit{min}} = \infty \\
4. \textbf{for } k = i \textbf{ to } j \\
5. \quad v = F(i, k - 1) + F(k + 1, j) \\
6. \quad \textbf{if } v < \text{\textit{min}} \textbf{ then } \text{\textit{min}} = v \\
7. \textbf{return } \text{\textit{common}} + \text{\textit{min}}
Naive Recursion

The algorithm in the previous slide is extremely expensive — its running time is $\Omega(3^n)!$

The crucial reason behind the inefficiency is that it does plenty of wasteful computation: e.g., if you run $F(1, 4)$, you will see that the algorithm computes $F(2, 2)$ repeatedly for 5 times!

This is a typical scenario that can be dealt with using the dynamic programming technique. Its objective is to avoid as much as possible re-computation by memorizing the $F(i, j)$ values that have already been computed.
The “Matrix View” of Dynamic Programming

Let us take a different approach to compute $F(i, j)$. Treat $F$ as an $n \times n$ matrix.

Our goal is to fill in all the cells of the matrix. We will do so by processing the cells in “groups”:

Define the **group number** of cell $F(i, j)$ as $j - i$. A **group** consists of all the cells with the same group number.

Note that all the cells with **negative** group numbers will be filled with 0 for sure.
The “Matrix View” of Dynamic Programming

Lemma: Consider cell $F(i, j)$; denote by $g = j - i$ its group number. Suppose that all the cells of group number $g - 1$ have been properly filled. Then, we can fill in $F(i, j)$ in $O(n)$ time.

Proof: Follows directly from the recurrence

$$F(i, j) = \left( \sum_{k=i}^{j} A[k] \right) + \min_{k=i} \left\{ F(i, k - 1) + F(k + 1, j) \right\}$$

noticing that each $F(i, k - 1)$ and $F(k + 1, j)$ can be obtained in $O(1)$ time.
algorithm Fill-$F$
1. fill all cells $F(i,j)$ satisfying $n \geq i > j \geq 1$ with 0
2. for $g = 0$ to $n - 1$
   /* $g$ is the group number */
3. for every cell $F(i,j)$ satisfying $j - i = g$
4. apply the lemma of Slide 8 to compute $F(i,j)$
Example: Suppose that $A = (40, 15, 35, 10)$
We fill the cells of $F$ in the following order:

- Cells with negative group numbers:
  Set $F(i, j) = 0$ for all $i, j$ satisfying $i > j$

- Cells of Group 0:
  $F(1, 1) = 40$, $F(2, 2) = 15$, $F(3, 3) = 35$, $F(4, 4) = 10$

- Cells of Group 1:
  $F(1, 2) = 70$, $F(2, 3) = 65$, $F(3, 4) = 55$

- Cells of Group 2:
  $F(1, 3) = 155$, $F(2, 4) = 85$

- The only cell with group number 3: $F(1, 4) = 180$
Now let us analyze the running time of the algorithm in Slide 9.

Line 1 clearly takes $O(n^2)$ time.
The for-loop at Lines 2-4 runs for $n$ times.
The for-loop at Lines 3-4 runs for at most $n$ times (each group has at most $n$ cells).
Line 4 takes $O(n)$ time.

Therefore, overall the algorithm runs in $O(n^3)$ time.
The above problem, in spite of its simplicity, illustrates adequately the rationales behind the dynamic programming technique. Recall that, by solving the problem recursively in a straightforward manner, we ended up with an exponential time complexity. Dynamic programming lowered the complexity to a polynomial function by memorizing the key information already computed, thus avoiding the need to recompute the same information again and again.