Design & Analysis of Algorithms (Dynamic Programming)

Dr. Soharab Hossain Shaikh
BML Munjal University

1

Dynamic Programming

Dynamic programming is both a mathematical optimization method and a computer programming method. The method was developed by Richard Bellman in the 1950s and has found applications in numerous fields, from aerospace engineering to economics.

Richard Ernest Bellman (August 26, 1920 – March 19, 1984) was an American applied mathematician, who introduced dynamic programming in 1953, and made important contributions in other fields of mathematics.

2

Multistage Optimization

Bellman stated the principle of optimality which explains the process of multi-stage optimization as:

"An optimal policy (or a set of decisions) has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."

3

Key Concepts in DP

- Complex problems are sometimes solved quickly if approached in a sequential manner
- Dynamic Programming : Sequential or multistage decision-making process
- Basic approach behind dynamic programming: Solution is found out in multi stages
- Works in a "divide and conquer" manner.

4

Dynamic Programming: Introduction

Dynamic Programming refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a **recursive** manner.

Principle of Optimality: A problem that can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems is said to have **optimal substructure**.

> In the bottom-up approach, solutions to the sub-problems are found and stored in a Table for future use.

> Solutions to larger problems are found by **reusing** the values stored in the table **without re-computing**.

> The word "**Programming**" refers to **planning** or tabulating the results of sub-problems into the table for future use.

> Used for Optimization problems.

5

Principle of Optimality

- Definition: A problem is said to satisfy the Principle of Optimality if the sub-solutions of an optimal solution of the problem are themselves optimal solutions for their subproblems.
- Examples:
  - The shortest path problem satisfies the Principle of Optimality.
  - This is because if *a,x1,x2,...,xn,b* is a shortest path from node *a* to node *b* in a graph, then the portion of *xi* to *xj* on that path is a shortest path from *xi* to *xj*.

6

Sequential Optimization

- Problem is divided into smaller sub-problems
- Optimize these sub-problems without losing the integrity of the original problem
- Decisions are made sequentially
- Also called multistage decision problems since decisions are made at a number of stages
- A N variable problem is represented by N single variable problems
- These problems are solved successively to get the optimal value of the original problem

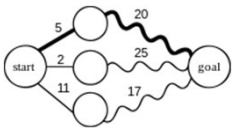
7

Optimal Substructure Property

- **In computer science, a problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems**.
- This property is used to determine the usefulness of dynamic programming and greedy algorithms for a problem.
- Typically, a greedy algorithm is used to solve a problem with optimal substructure if it can be proven by induction that this is optimal at each step.
- Otherwise, provided the problem exhibits overlapping subproblems as well, dynamic programming is used.
- If there are no appropriate greedy algorithms and the problem fails to exhibit overlapping subproblems, often a lengthy but straightforward search of the solution space is the best alternative.
- In the application of dynamic programming to mathematical optimization, **Richard Bellman**'s Principle of Optimality is based on the idea that in order to solve a dynamic optimization problem from some starting period t to some ending period T, one implicitly has to solve subproblems starting from later dates s, where t<s<T. This is an example of optimal substructure.

8

Optimal Substructure Property



Finding the shortest path using optimal substructure.

Numbers represent the length of the path;
Straight lines indicate single edges;
Wavy lines indicate shortest paths, i.e., there might be other vertices that are not shown here.

- Consider finding a **shortest path** for travelling between two cities by car, as illustrated in the figure.
- Such an example is likely to exhibit optimal substructure. That is, if the shortest route from Seattle to Los Angeles passes through Portland and then Sacramento, then the shortest route from Portland to Los Angeles must pass through Sacramento too.
- That is, the problem of how to get from Portland to Los Angeles is nested inside the problem of how to get from Seattle to Los Angeles. (The wavy lines in the graph represent solutions to the subproblems.)

9

Problem *without* Optimal Substructure

- **Longest Path Problem**
- *Least-cost airline fare.* (Using online flight search, we will frequently find that the cheapest flight from airport A to airport B involves a single connection through airport C, but the cheapest flight from airport A to airport C involves a connection through some other airport D.).
- Real example - finding the cheapest airline ticket from Buenos Aires to Moscow. Even if that ticket involves stops in Miami and then London, we can't conclude that the cheapest ticket from Miami to Moscow stops in London, because the price at which an airline sells a multi-flight trip is usually not the sum of the prices at which it would sell the individual flights in the trip.
- However, if the problem takes the maximum number of layovers as a parameter, then the problem has optimal substructure: the cheapest flight from A to B involving a single connection is either the direct flight; or a flight from A to some destination C, plus the optimum direct flight from C to B.

10

Greedy vs. Dynamic Programming

- Both techniques are optimization techniques, and both build solutions from a collection of choices of individual elements.
- The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.
- Dynamic programming computes its solution bottom up by synthesizing them from smaller sub-solutions, and by trying many possibilities and choices before it arrives at the optimal set of choices.
- There is no apriori test by which one can tell if the Greedy method will lead to an optimal solution. By contrast, there is a test for Dynamic Programming, called **The Principle of Optimality**.

11

D&C vs. Dynamic Programming

- **Divide and Conquer vs. Dynamic Programming**
- Both techniques split their input into parts, find sub-solutions to the parts, and synthesize larger solutions from smaller ones.
- Divide and Conquer splits its input at prespecified deterministic points (e.g., always in the middle)
- Dynamic Programming splits its input at every possible split points rather than at a pre-specified points. After trying all split points, it determines which split point is optimal.

12

Computing the n-th Fibonacci Number

13

Dynamic Programming: Introduction

- Dynamic programming algorithms address problems whose solutions are recursive in nature with the following property: The direct implementation of a recursive solution results in *identical recursive calls* that are executed *more than once*.
- **Example 1**: Design an algorithm that computes the nth Fibonacci number:

$$f_n = \begin{cases} 1 & n = 1 \text{ or } n = 2 \\ f_{n-1} + f_{n-2} & n \geq 3 \end{cases}$$

A direct recursive algorithm that computes f_n is as follows:

Procedure $f(n)$
1. **if** $(n = 1)$ **or** $(n = 2)$ **then return** 1
2. **else return** $f(n-1) + f(n-2)$

Expanding the computation of $f(10)$ for a few terms, we get

$f(10) = f(9) + f(8)$
$= f(8) + f(7) + f(7) + f(6)$
$= f(7) + f(6) + f(6) + f(5) + f(6) + f(5) + f(5) + f(4)$

14

Dynamic Programming: Introduction (cont…)

- **Example 1** (Cont.):
  - Time complexity of $f(n)$

Procedure $f(n)$
1. **if** $(n = 1)$ **or** $(n = 2)$ **then return** 1
2. **else return** $f(n-1) + f(n-2)$

$$T(n) = \begin{cases} 1 & n = 1 \text{ or } n = 2 \\ T(n-1) + T(n-2) & n \geq 3 \end{cases}$$

Clearly, $T(n) = f_n$.

Since T (n) is a homogeneous equation, and by introducing the value $T(0) = 0$, it can be shown by means of the characteristic equation that $f_n = \Theta(\phi^n)$ where ϕ is the golden ratio $\frac{1+\sqrt{5}}{2} \approx 1.61803$.

Hence, $f(n)$ has an exponential time complexity in the value of n.

15

Dynamic Programming: Introduction (cont…)

- **Example 1** (Cont.):
  Consider the following procedure that computes the n^{th} Fibonacci number:

Procedure *fib*(n)
Comment $A[1..n]$ will hold the values of f_n
1. $A[1] \leftarrow 1$
2. $A[2] \leftarrow 1$
3. **for** $i \leftarrow 3$ to n **do**
4. $\quad A[i] \leftarrow A[i\text{-}1] + A[i\text{-}2]$
5. **end for**
6. return $A[n]$;

  - Procedure *fib* runs in $\Theta(n)$ time, a significant improvement over Procedure f.
  - Procedure *fib* requires $\Theta(n)$ space, unlike Procedure f which requires $\Theta(1)$ space.
  - The saving in time complexity from an exponential algorithm to a linear time algorithm justifies the polynomial increase in space complexity.

16

Longest Common Subsequence (LCS) Problem

17

Longest Common Subsequence Problem

- **Definition**: A subsequence of a string $A = a_1 a_2 ... a_n$ is a string of the form $a_{i_1} a_{i_2} ... a_{i_k}$ where $1 \le i_1 < i_2 < ... < i_k \le n$
  - Note: If $i_1, i_2, ..., i_k$ are consecutive, the subsequence will be called a substring.
- **Example 1**: Let $A = xyzyxxyz$. Then
  - $S_1 = xyzy$ is a subsequence of A that is a substring.
  - $S_2 = xxxz$ is another subsequence of A that is not a substring
  - $S_3 = zzx$ is neither a subsequence nor a substring of A

18

LCS (cont…)

- **Problem Definition**: Given two strings A and B of lengths m and n, respectively, over an alphabet Σ, determine the length of the longest subsequence that is common in A and B.
- **Example 2**: Let $\Sigma = \{x, y, z\}$ and suppose that we have the following two strings:

  $A = xyxyxxzy$ and $B = yxyyzxy$

  - $yxyy$ is a common subsequence in A and B. However, it is not a longest common subsequence.
  - $xyyzy$ is a longest common subsequence of A and B.
  - $yxyzy$ is another longest common subsequence of A and B.
- The longest common subsequence of two strings may not be unique.

19

LCS: Brute Force Solution

- List all subsequences of one string, ordered in non-increasing order of their length, and then return the length of the first subsequence that is also a subsequence in the other string.
- Time complexity analysis of this solution
  - Given two strings A and B of lengths m and n, respectively, where $m \le n$, there exist 2^m different subsequences in A.
  - Each subsequence in A requires $\Theta(n)$ comparisons in order to verify whether it is a subsequence of B or not.
  - The total time complexity of the brute-force search method is $O(2^m n)$, which is exponential in input size.

20

LCS: Recursive Solution

- Recursive Solution for the LCS Problem
  - Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ be two strings, and let $L[i, j]$ denote the length of the longest common subsequence of $a_1 a_2 \ldots a_i$ and $b_1 b_2 \ldots b_j$. An empty string is represented by having i equals zero or j equals zero.

$$L(i,j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ L(i-1, j-1) + 1 & a_i = b_j, i > 0 \text{ and } j > 0 \\ \max\{L(i-1, j), L(i, j-1)\} & a_i \neq b_j, i > 0 \text{ and } j > 0 \end{cases}$$

- Bottom-up Computation of $L[i, j]$
  - Use an $(m+1) \times (n+1)$ table to compute the values of $L[i, j]$ using the above recurrence row by row, where $0 \le i \le m$ and $0 \le j \le n$.

21

LCS: Algorithm

Algorithm LCS
Input: Two strings A and B of lengths m and n, respectively, over an alphabet Σ.
Output: The length of the longest common subsequence of A and B.

1. **for** $i \leftarrow 0$ **to** m **do**
2. $L[i, 0] \leftarrow 0$
3. **end for**
4. **for** $j \leftarrow 0$ **to** n **do**
5. $L[0, j] \leftarrow 0$
6. **end for**
7. **for** $i \leftarrow 1$ **to** m **do**
8. **for** $j \leftarrow 1$ **to** n **do**
9. **if** $a_i = b_j$ **then** $L[i, j] \leftarrow L[i-1, j-1] + 1$
10. **else** $L[i, j] \leftarrow \max\{L[i-1, j], L[i, j-1]\}$
11. **end if**
12. **end for**
13. **end for**
14. **return** $L[n, m]$

Complexity Analysis of LCS

Time Complexity: $\Theta(mn)$
Space Complexity: $\Theta(mn)$

22

Example: LCS

Example 3: Use Algorithm LCS to find the length of an LCS of the two strings $A = xzyzzyx$ and $B = zxyyzxz$.

		x	z	y	z	z	y	x
	0	0	0	0	0	0	0	0
z	0	0	1	1	1	1	1	1
x	0	1	1	1	1	1	1	2
y	0	1	1	2	2	2	2	2
y	0	1	1	2	2	2	3	3
z	0	1	2	2	3	3	3	3
x	0	1	2	2	3	3	3	4
z	0	1	2	2	3	4	4	4

23

LCS: Algorithm (Revised)

Length + the actual Sequence

```
LCS(char x[1..m], char y[1..n]) {
    int c[0..m, 0..n]
    for i = 0 to m do {
        c[i,0] = 0    b[i,0] = SKIPX              // initialize column 0
    }
    for j = 0 to n do {
        c[0,j] = 0    b[0,j] = SKIPY              // initialize row 0
    }
    for i = 1 to m do {
        for j = 1 to n do {
            if (x[i] == y[j]) {
                c[i,j] = c[i-1,j-1]+1             // take X[i] and Y[j] for LCS
                b[i,j] = ADDXY
            }
            else if (c[i-1,j] >= c[i,j-1]) {      // X[i] not in LCS
                c[i,j] = c[i-1,j]
                b[i,j] = SKIPX
            }
            else {                                // Y[j] not in LCS

                    c[i,j] = c[i,j-1]
                    b[i,j] = SKIPY
                }
            }
        }
        return c[m,n];
    }
```

24

Extract the Actual Sequence

Extracting the Actual Sequence: Extracting the final LCS is done by using the back pointers stored in $b[0..m, 0..n]$. Intuitively $b[i,j] = ADDXY$ means that $X[i]$ and $Y[j]$ together form the last character of the LCS. So we take this common character, and continue with entry $b[i-1, j-1]$ to the northwest ($\nwarrow$). If $b[i,j] = SKIPX$, then we know that $X[i]$ is not in the LCS, and so we skip it and go to $b[i-1, j]$ above us ($\uparrow$). Similarly, if $b[i,j] = SKIPY$, then we know that $Y[j]$ is not in the LCS, and so we skip it and go to $b[i, j-1]$ to the left ($\leftarrow$). Following these back pointers, and outputting a character with each diagonal move gives the final subsequence.

25

Extract the Actual Sequence

```
getLCS(char x[1..m], char y[1..n], int b[0..m,0..n]) {
    LCS = empty string
    i = m
    j = n
    while(i != 0 && j != 0) {
        switch b[i,j] {
        case ADDXY:
            add x[i] (or equivalently y[j]) to front of LCS
            i--;   j--;   break
        case SKIPX:
            i--;  break
        case SKIPY:
            j--;  break
        }
    }
    return LCS
}
```

26

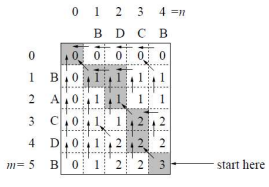
Example: LCS

		0	1	2	3	4 = n
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
m = 5	B	0	1	2	2	3

LCS Length Table

X = BACDB
Y = BDCB

LCS = BCB



with back pointers included

27

End of Lecture

28