CS 491 CAP
Intro to Dynamic Programming

Jingbo Shang

University of Illinois at Urbana-Champaign

Sept 29, 2017

Today

- ◊ What is DP?
- ◊ 3 example problems
  - Introductory example
  - Longest Common Subsequence
  - Coin Change

What is Dynamic Programming?

- Algorithm design technique/paradigm
- "Method for solving complex problems by breaking them down into smaller subproblems" - Wikipedia
- This definition will make more sense once we see some concrete problems

Prerequisites

◊ Familiar with recursion

illinois.edu

Example 1

- Given an array A of N integers
- You want to pick some elements from the array
- However, no two picked elements can be adjacent
- How can you pick the elements so that their sum is maximized, while satisfying the constraint?
- Compute the optimal sum you can obtain
- Input: A = [7, 1, 5, 8, 2]
- Answer: 15 (pick 7 and 8)

Recursive solution

- ◇ Consider the last element, A[N]
- ◇ Any solution will either contain A[N] or not
- ◇ Case 1) If a solution contains A[N]
  - Then A[N - 1] cannot be in the solution, so we can reduce the problem into solving for A[1 .. N - 2] instead and add A[N] after
- ◇ Case 2) If a solution doesn't contain A[N]
  - No restriction on A[N - 1], so we can reduce the problem into solving for A[1 .. N - 1]
- ◇ Take the maximum of these two cases
- ◇ Base case:
  - N = 1, return A[1]
  - N = 2, return max(A[1], A[2])

Recursive Solution - Pseudocode

```
procedure solve
    input: A[1...N], an array of integers
    output: the optimal sum that does not
            contain adjacent numbers

    if N == 1:
        return A[1]
    if N == 2:
        return max(A[1], A[2])
    return max(solve(A[1...N-1],
                solve(A[1...N-2] + A[N])
```

Too slow…

- ◇ This solution is exponential!
  - We compute the same subproblem multiple times
- ◇ How can we improve?
- ◇ Each subproblem can be represented as the last index of the subarray
  - Take the array out of the parameter and represent with an integer instead
- ◇ If we have already computed the solution for a subproblem, just return the computed value
- ◇ Save the computed solution to the subproblem in a table
- ◇ This technique is called “memoization”

Recursive Solution with Memoization

```
use D[1...N] to store the results of the subproblems,
  mark them as not computed initially

procedure solve
    input: A[1...N], an array of integers
    output: the optimal sum that does not
            contain adjacent numbers

    if D[N] is not computed:
        if N == 1:
            D[N] = A[1]
        if N == 2:
            D[N] = max(A[1], A[2])
        else
            D[N] = max(solve(A[1...N-1],
                       solve(A[1...N-2] + A[N])
    return D[N]
```

illinois.edu

Iterative Solution

- The improved recursive solution is O(N)
- We can improve the speed (a bit) by translating the recursive solution to an iterative solution
  - Asymptotic running time is still O(N)
  - But we get rid of recursion overhead, etc…
  - However downsides exist, will address later

Iterative Solution - Pseudocode

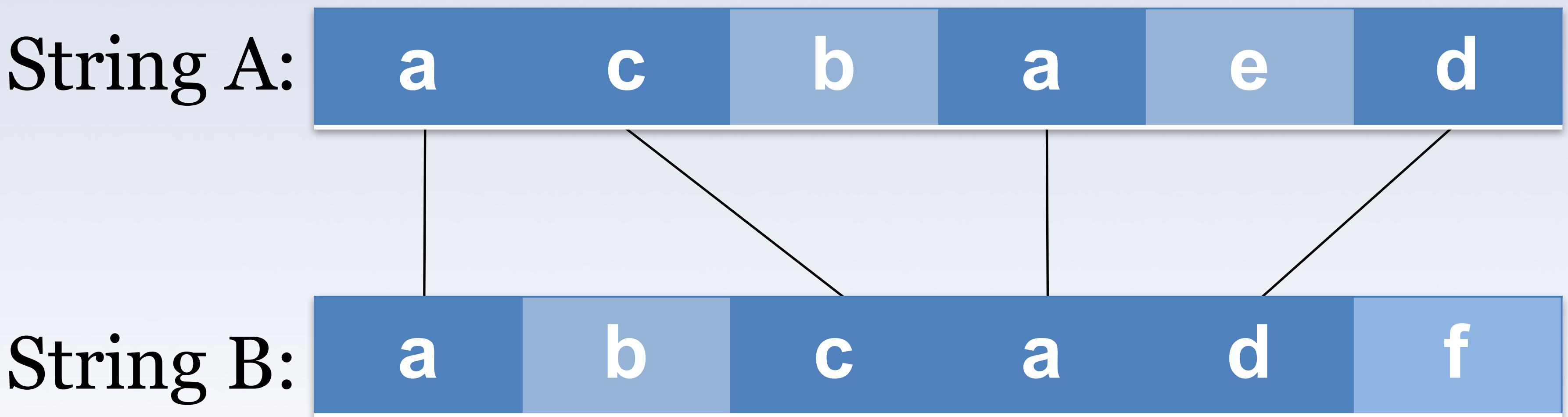
```
procedure solve
    input: A[1...N], an array of integers
    output: the optimal sum that does not
            contain adjacent numbers

    D[1] = A[1]
    D[2] = max(A[1], A[2])
    for i = 3 to N:
        D[i] = max(D[i - 1], A[i] + D[i - 2])
    return D[N]
```

illinois.edu

Example 2

- You are given two strings A and B of length N and M
- Compute the length of the longest common subsequence of A and B
- A subsequence of a string S is defined as $S[i_1]S[i_2]...S[i_n]$ such that $1 <= i_1 < i_2 < ... < i_n <= len(S)$
  - Intuitively, a string that you can obtain after deleting some number of characters from the string
- A longest common subsequence of two strings A and B is the longest subsequence that appears in both A and B

LCS (Longest Common Subsequence) Example



Recursive solution

- ◊ Consider A[N] and B[M] (the last characters)
- ◊ Two cases, A[N] = B[M] or A[N] != B[M]
- ◊ Case 1) A[N] = B[M]:
  - Easy to see that A[N] (or B[M]) is the last character of the desired LCS, so we can reduce the problem into computing LCS length of A[1 .. N - 1] and B[1 .. M - 1]
  - Add 1 to the recursively computed LCS length

Recursive solution contd

- Case 2) A[N] != B[M]:
  - Clearly, any desired LCS must be in A[1 .. N] and B[1 .. M - 1] OR in A[1 .. N - 1] and B[1 .. M]
  - Reduce the problem into computing LCS length of A[1 .. N] and B[1 .. M - 1] and of A[1 .. N - 1] and B[1 .. M]
  - Take the maximum of these two subcases
- Base case: if either of the strings is empty, then clearly the length of the LCS is 0

Recursive Solution - Pseudocode

```
algorithm lcs
    input: two strings, A[1...N], B[1...M]
    output: the length of the longest common
            subsequence of the two strings

    if either A or B is empty:
        return 0
    if A[N] == B[M]:
        return lcs(A[1...N-1], B[1...M-1]) + 1
    else
        return max(lcs(A[1...N-1], B[1...M]
                   lcs(A[1...N], B[1...M-1])
```

Recursive Solution With Memoization

```
use D[0...N][0...M] to store the results of the subproblems,
  mark the whole array as not computed

algorithm lcs
    input: two strings, A[1...N], B[1...M]
    output: the length of the longest common
            subsequence of the two strings

    if D[N][M] is not computed:
        if either A or B is empty:
            D[N][M] = 0
        if A[N] == B[M]:
            D[N][M] = lcs(A[1...N-1], B[1...M-1]) + 1
        else
            D[N][M] = max(lcs(A[1...N-1], B[1...M]
                          lcs(A[1...N], B[1...M-1])
    return D[N][M]
```

Iterative Solution

```
algorithm lcs
    input: two strings, A[1...N], B[1...M]
    output: the length of the longest common
            subsequence of the two strings

    for i = 0 to N:
        D[i][0] = 0
    for i = 0 to M:
        D[0][i] = 0
    for i = 1 to N:
        for j = 1 to M:
            if A[i] == B[j]:
                D[i][j] = D[i-1][j-1] + 1
            else
                D[i][j] = max(D[i-1][j], D[i][j-1])
    return D[N][M]
```

LCS Running time

- The running time of this algorithm is O(NM)

More on Dynamic Programming

- So far, the problems you've seen asked to maximize/minimize some quantity
- These are examples of optimization problems
- DP is also widely used in solving combinatorics problems
- i.e. Count the number of ways to do something, compute the probability, etc
- We'll solve a simple combinatorics problem today

Example 3

- Given an amount N cents
- Given a set of coin types C, each type with infinite amount
- Count the number of ways to make N cents
- Example:
  - N = 5
  - C = [1, 2, 3]
  - Answer: 5
    - 1, 1, 1, 1, 1
    - 1, 1, 1, 2
    - 1, 1, 3
    - 1, 2, 2
    - 2, 3

Recursive solution

- ◇ Suppose C consists of M coin types
- ◇ Consider C[M], the Mth coin type
- ◇ Two cases: Mth coin type is never used, or is used at least once
  - If the Mth coin type is never used, then the problem reduces to making N cents with the first M - 1 coin types
  - If the Mth coin type is used at least once, then the problem reduces to making N - C[M] cents with the all M coin types
    - N - C[M] enforces that Mth coin type is used at least once
- ◇ These two cases are clearly disjoint, so sum these two cases to get the desired result

Recursive Solution - Pseudocode

```
procedure solve
    input:  N, the number of cents
            C[1...M], the value of each type of coin
    output: the number of ways to make N cents

    if N == 0:
        return 1
    if N < 0 or M = 0:
        return 0
    return solve(N, M - 1) + solve(N - C[M], M)
```

Can be easily optimized with memoization.
Time Complexity: O(NM)

illinois.edu

Recursive vs Iterative Revisited

- Which one is better?
- Iterative solutions are usually shorter, faster by some constant time
- However iterative methods do require the coder to work out the order the subproblems are computed, which could become harder and harder in higher dimensions
- Iterative solutions can be easily optimized with memoization and we usually don't really care about the overheads

Questions?

- Memoization/iterative solution for the last problem left as an exercise
  - Can you reduce the space complexity to O(N) instead of O(NM)?
- We will cover more advanced topics in DP few weeks later