

Dynamic Programming

CS 5633 Analysis of Algorithms

Computer Science
University of Texas at San Antonio

October 27, 2024

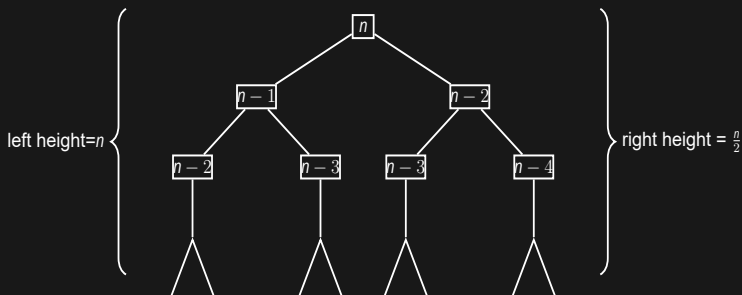
Divide and Conquer with Reusable Sub-problems

Fibonacci Numbers with Divide and Conquer

- ▶ So far in class, the problems which we have solved with a divide-and-conquer algorithm have had the property that a subproblem we recursively generate will never be generated more than once.
 - For example in Merge-Sort($A[1..n]$), we recursively call Merge-Sort($A[1..n/2]$) and Merge-Sort($A[n/2 + 1..n]$). At no other point in time of the algorithm will we again make one of these calls.
- ▶ There are some other problems which have natural recursive definitions for which this is not the case.
- ▶ For example the *Fibonacci Sequence*:
 - 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
 - $F(1) = 1$, $F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ for $n \geq 3$.

Recursive Tree of Fibonacci Numbers

- Recursion tree for computing the n th Fibonacci number $F(n)$:
 - $F(1) = 1, F(2) = 1, F(n) = F(n-1) + F(n-2)$ for $n \geq 3$.



- Run time $T(n)$ is the total number of nodes.
 - Based on binary tree properties,
 $2^{n/2} \leq T(n) \leq 2^n \implies T(n) = \Theta(2^n)$.

Improving the Fibonacci Algorithm

- ▶ The running time of the straight-forward recursive algorithm for computing $F(n)$ takes $O(2^n)$ time, but we compute the same sub problems multiple times.
- ▶ Idea: check to see if we have already solved a subproblem. If we have not solved it yet, then do so now and store the value of the subproblem in an array. If we have already solved it, then the value is stored in the array and we can return the value of the subproblem in $O(1)$ time rather than making another recursive call.
- ▶ This technique is called **memoization** and is a variant of **dynamic programming**.

Memoization Algorithm for Fibonacci Numbers

- Memoization algorithm for the Fibonacci sequence. F is an array of size n with each element initialized to null.

Algorithm 1: Memoization Algorithm for Fibonacci Numbers

```
1 Function fib_memoization(int  $n$ , Array  $F$ )
2   if  $F[n] \neq \text{NULL}$  then
3     return  $F[n]$ ; // fib( $n$ ) is calculated;
4   // fib( $n$ ) is not calculated yet;
5   if  $n == 1$  OR  $n == 2$  then
6      $F[n] = 1$ ;
7   else
8     // recursively compute fib( $n$ ) ;
9      $F[n] = \text{fib\_memoization}(n - 1, F) + \text{fib\_memoization}(n - 2, F)$ ;
10  return  $F[n]$ ;
```

Run-time Memoization Algorithm for Fibonacci Numbers

- ▶ Clearly, the memoization algorithm is essentially filling the n element array F .
- ▶ Therefore, the time complexity is $\Theta(n)$.
- ▶ The algorithm also requires extra space for F . The space cost (size of the array) is $\Theta(n)$.

Top-down vs Bottom-up

- ▶ Memoization is a “top-down” approach in which we start at the top of the recursion tree as we have been doing and work our way down the tree as is done in a regular recursive algorithm.
- ▶ An alternative is a “bottom-up” approach in which we first start by computing the smallest subproblems (i.e. the base case) and then compute the larger subproblems based off of the smaller subproblems which we already solved.
- ▶ This is typically done with an iterative algorithm (using loops instead of recursion). This is the other variant of dynamic programming.

A Bottom-up Algorithm for Fibonacci Numbers

- ▶ The bottom-up algorithm is essentially a loop iterating over F to fill its values one by one.

Algorithm 2: Bottom Up Memonization Algorithm for Finbonacci Numbers

```
1 Function fib_BottomUp(int  $n$ , Array  $F$ )  
2    $F[1] = F[2] = 1$ ;  
3   for  $i \leftarrow 3$  to  $n$  do  
4      $F[i] = F[i - 1] + F[i - 2]$ ;  
5   return  $F[n]$ ;
```

- ▶ Clearly, the time and space cost of this algorithm are both $\Theta(n)$.

Two-dimensional Dynamic Programming

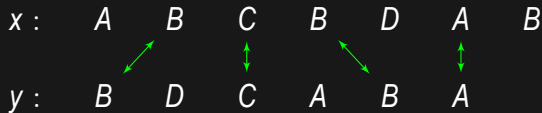
The Longest Common Subsequence (LCS) Problem

- ▶ The Fibonacci number problem is a one-dimensional dynamic programming problem.
 - The sub-problem space is linear, i.e, sub problems are identified with a single number n .
- ▶ Most common dynamic program problems has multi-dimensional sub-problem space.
- ▶ For example, let's consider this problem:
 - Given two sequences of characters $x[1 : m]$ and $y[1 : n]$, find a longest subsequence common to them both (note that a subsequence and a substring are different).

An Example of LCS problem

- Considering the following two strings x and y :

x :	A	B	C	B	D	A	B
y :	B	D	C	A	B	A	



- The longest common sequence is:
 $LCS(x, y) = \{BCBA\}$.

Brute-for LCS Algorithm

- ▶ Consider a brute-force algorithm for computing LCS.
 - Check every subsequence of $x[1..m]$ and check to see if it is also a subsequence of $y[1..n]$.
- ▶ How many subsequences does x contain?
 - This is essentially asking how many subsets does x has?
 - 2^m (each 0/1 vector of length m determines a distinct subsequence of x).
- ▶ This algorithm would have exponential running time.
 - 2^m subsets of $x \times 2^n$ subsets of y .

Recursively Solving LCS

- ▶ Can we break down a LCS problem into sub problems?
- ▶ Considering these two strings again:

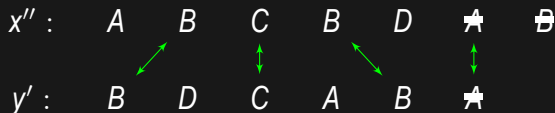
x : A B C B D A B
 ↗ ↘ ↕ ↗ ↘ ↕
 y : B D C A B A

- ▶ If we remove the last character from x , the result is the same.

x' : A B C B D A ~~B~~
 ↗ ↘ ↕ ↗ ↘ ↕
 y : B D C A B A

Recursively Solving LCS cont.

- ▶ Considering now the last letters of x and y are the same, we can remove the last A from both string, and compute the LCS of the new strings. The LCS for the origin strings will be the LCS of the new strings plus A .



- ▶ Essentially, $LCS(x, y) = LCS(x', y) = LCS(x'', y') + A$.

Recursively Solving LCS cont.

- ▶ If we summarize the above examples:
 - Case 1: If $x[m] == y[n]$, we have
$$LCS(x[1 : m], y[1 : n]) = LCS(x[1 : m - 1], y[1 : n - 1]) + 1.$$
 - Case 2: If $x[m] \neq y[n]$, we have $LCS(x[1 : m], y[1 : n])$ be the larger one of
 - ▶ $LCS(x[1 : m - 1], y[1 : n])$ or,
 - ▶ $LCS(x[1 : m], y[1 : n - 1])$

The Recursive LCS Algorithm



Algorithm 3: The Recursive LCS Algorithm

```
1 Function LCS_Recursive( $x[1 : m]$ ,  $y[1 : n]$ )
2   if  $m == 0$  OR  $y == 0$  then
3     return []; // LCS is empty ;
4   if  $x[m] == y[n]$  then
5     return LCS_Recursive( $x[1 : m - 1]$ ,  $y[1 : n - 1]$ ) +  $x[m]$ ; // Case 1;
6   else
7     // Case 2;
8      $LCS1 = \text{LCS\_Recursive}(x[1 : m - 1], y[1 : n]);$ 
9      $LCS2 = \text{LCS\_Recursive}(x[1 : m], y[1 : n - 1]);$ 
10    return  $\text{MAX}(LCS1, LCS2);$ 
```

Run Time of The Recursive LCS Algorithm

- ▶ This algorithm, unfortunately, still takes exponential time to execute.
- ▶ Getting the LCS of $(x[1 : m - 1], y[1 : n])$ and the LCS of $(x[1 : m], y[1 : n - 1])$ will eventually process same pairs of strings.
 - For example, they both will try to find the LCS of $(x[1 : 1], y[1 : 1])$, and many other same pairs.

Constructing Matrix for Dynamic Programming LCS Algorithm

- ▶ To reuse the results of sub problems, let's define a 2D matrix S .
 - $S[i, j]$ represents the LCS of $x[1 : i]$ and $y[1 : j]$.
 - $S[0, *]$ and $S[*, 0]$ are empty sequences. These items are use as bases for matrix construction.
- ▶ S can be constructed similarly with the recursive algorithm.
 - Case 1: If $x[i] == y[j]$, then $S[i, j] = S[i - 1, j - 1] + x[i]$.
 - Case 2: If $x[i] \neq y[j]$, then $S[i, j]$ is the longest sequence of $S[i - 1, j]$ and $S[i, j - 1]$. If they have the same length, take either one is fine.

The Dynamic Programming LCS Algorithm

- We can then convert the recursive algorithm into dynamic programming algorithm.

Algorithm 4: The Dynamic Programming LCS Algorithm

```
1 Function LCS_DP( $x, i, y, j$ )
2   if  $S[i, j]$  is not NULL then
3     return  $S[i, j]$ ;
4   if  $i == 0$  OR  $j == 0$  then
5      $S[i, j] = []$ ; // LCS is empty ;
6   else if  $x[i] == y[j]$  then
7      $S[i, j] = \text{LCS\_DP}(x, i - 1, y, j - 1) + x[i]$ ; // Case 1;
8   else
9     // Case 2;
10     $S[i - 1, j] = \text{LCS\_DP}(x, i - 1, y, j)$ ;
11     $S[i, j - 1] = \text{LCS\_DP}(x, i, y, j - 1)$ ;
12     $S[i, j] = \text{MAX}(S[i - 1, j], S[i, j - 1])$ ;
13  return  $S[i, j]$ ;
```

Run Time of LCS_DP

- ▶ The algorithm essentially sets the value for each $S[i, j]$.
- ▶ Matrix S has a size of $m \times n$. Therefore, the time cost is $\Theta(mn)$.
- ▶ The space cost is $\Theta(m \times n \times \min(m, n))$
 - Because each cell of S may store a sequence of at most $\min(m, n)$ long.
- ▶ Similarly to fibonacci numbers, we can also construct S with a bottom-up algorithm.

Illustration of LCS_DP

- ▶ The array S constructed with LCS_DP for the previous example.
- ▶ Empty cells represent no LCS.

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i							
1	A					A	A	A
2	B		B	B	B	B	AB	AB
3	C		B	B	BC	BC	AB	AB
4	B		B	B	BC	BC	BCB	BCB
5	D		B	BD	BC	BC	BCB	BCB
6	A		B	BD	BD	BDA	BDA	BCBA
7	B		B	BD	BD	BDA	BDAB	BDAB

Steps to Design Dynamic Programming Algorithms

Steps to Design Dynamic Programming Algorithms

- ▶ The crucial part of dynamic programming is to find the common sub-problems that can be reused.
- ▶ In general, the steps for designing dynamic programming algorithms include,
 1. Break the original problem into sub-problems. The break down is most likely recursive.
 2. Manually run simple sub-problems to see if some of them can be reused.
 3. If yes, find a way to organize the sub-problems into arrays or matrices.
 4. Design algorithm base on the idea of reusing sub-problems.

Concerns for Dynamic Programming Algorithm Parallelization

- ▶ Dynamic programming algorithm is difficult to parallelize.
 - The central storage for storing solutions to sub-problems must be lock-protected, which is expensive.
 - A task is more likely to find the solution for one sub-problems.
 - There are many small tasks which require optimization to reduce task-spawn overhead.
 - Internal dependency can reduce the parallel efficiency of the algorithm, i.e., limiting the number of tasks that can run in parallel.
 - Tracking the dependency is also expensive.

Memoization in System Design

- ▶ Memoization is heavily utilized in compilers.
 - Compilers may allow user to specify a function memoization, especially in functional languages. So instead of doing a DP, just let the recursive function be memoized.
 - Run-time systems may automatically store the results from functions based on input parameters. They basically automatically make a recursive algorithm into dynamic programming algorithm.
 - Compilers use memoization to simplify code parsing.
- ▶ Memoization is an important Computer Science concept by itself.
 - Many numerical programs compute the same function with the same inputs over and over again. Memoization removes duplicated calls.
 - Function calls are expensive, memoization removes the cost to setup function calls.