SoC 2023: Competitive Programming
Week-5: Dynamic Programming

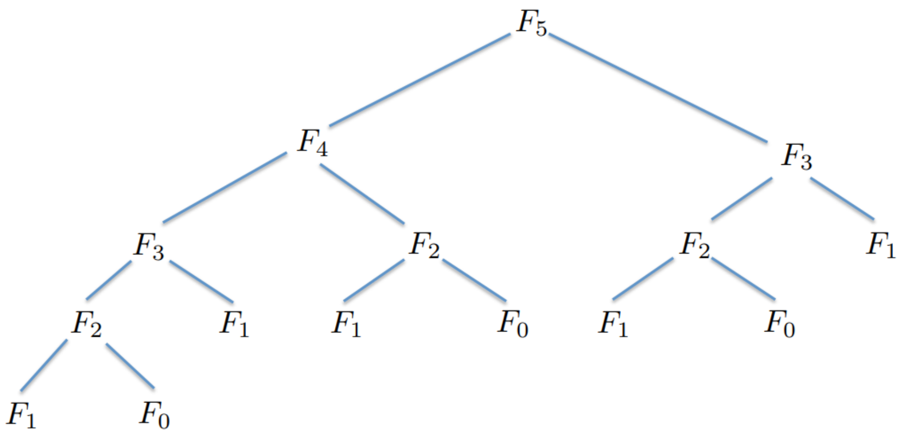
Mentor: Virendra Kabra

Summer 2023

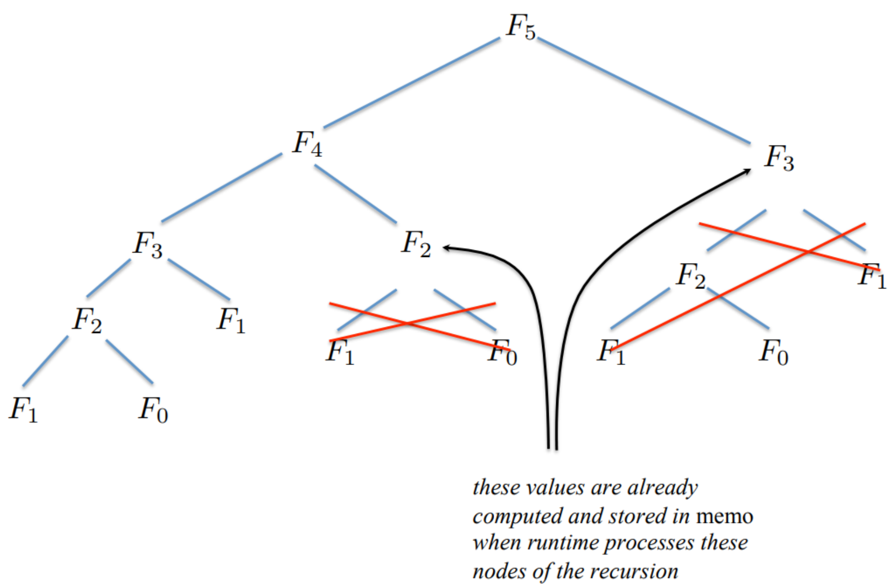
Contents

1 Basics

Dynamic Programming is similar to recursion - solving subproblems and combining them. In problems involving optimization (e.g., finding minimum cost to do a task), this is known as "optimal substructre" - optimal solutions to a problem involve optimal solutions to subproblems.
Computation of Fibonacci numbers: Image Ref



Memoization: Storing values of subproblems after computation is an important idea in DP. This helps when there is a reuse of values - "overlapping subproblems". Here, space of computed values is only $\{F_1, \ldots, F_n\}$, but the method has an exponential complexity, indicating that same values are being computed repeatedly.



Code: recursive and iterative.

2 Examples

- Binomial Coefficients. We saw a way to compute $\binom{n}{k} \pmod{p}$ earlier.
  - Recursion: $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$
  - Memoization: Observe that values are being used repeatedly. Two indices are involved, so we use a 2D matrix as memo. Can also use a `map<pair<int, int>, ll>` instead. Since the space of indices is known (and small), it is better (lower complexity) to use a matrix
  - Iterative version: Start with base cases, and fill up the matrix in order using recurrence relation.

n/k	**0**	**1**	**2**	**3**	**4**
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	5	10	10	5
6	1	6	15	20	15
7	1	7	21	35	35
8	1	8	28	56	70
9	1	9	36	84	126
10	1	10	45	120	210

  Code.

- Subset Sum. Given an array S of non-negative integers $\{a_1, \dots, a_n\}$ and a target value B, determine if there is a subset with sum B. Each number can be taken at most once.
  - Naïve method: Iterate over all subsets. Complexity: $O(n \cdot 2^n)$.
  - Recurrence: Make cases on the last element - it can/cannot be in the subset. This gives two subproblems.

  $$sol(S, B) = sol(S \setminus \{a_n\}, B - a_n) \mid\mid sol(S \setminus \{a_n\}, B)$$

  For implementation, only set indices are considered:

  $$sol(n, B) = sol(n-1, B - a_n) \mid\mid sol(n-1, B)$$

  - Again, subproblems repeat. Memoize using 2D array.

- 0-1 Knapsack Problem. Given weights and prices of n items ($\{w_1, \dots, w_n\}$ and $\{p_1, \dots, p_n\}$ (all non-negative)), put items in a knapsack of capacity W to get the maximum total value in the knapsack.
  - Similar to subset sum, last item can/cannot be included in the final subset.

  $$maxval(n, W) = \max(maxval(n-1, W - w_n) + p_n, maxval(n-1, W))$$

  - Memoize with a 2D array.

  Code.

- Unbounded Knapsack. Reference.

- Longest Increasing Subsequence (LIS). Given an array of size n, find the length of the longest subsequence such that all elements of the subsequence are in increasing order. For example, the length of LIS for $\{10, 22, 9, 33, 21, 50, 41, 60, 80\}$ is 6 ($\{10, 22, 33, 50, 60, 80\}$ and others).
  - Recurrence: Define $LIS(i)$ to be length of LIS from $[0, i]$. Then

$$LIS(i) = \max_{j<i \text{ and } a_j \le a_i} LIS(j) + 1$$

- Matrix Parenthesization.
  - Define the cost of multiplying two matrices A $(m \times n)$ and B $(n \times p)$ be $m \cdot n \cdot p$. Cost can be thought of as the number of operations.

    Given multiplication-compatible matrices $\{A_1, \ldots, A_n\}$ with dimensions $\{d_1, d_2, \ldots, d_n, d_{n+1}\}$, find a parenthesization so that cost is minimized.

    Example: A_1, A_2, A_3 with sizes $10 \times 100, 100 \times 5, 5 \times 50$.

    1. $(A_1 A_2)A_3$ has cost $5000 + 2500 = 7500$
    2. $A_1(A_2 A_3)$ has cost $25000 + 50000 = 75000$
  - Recursion: Subproblems for $A_i \ldots A_j$ are $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$.

$$cost(i, j) = \min_{i \le k < j} cost(i, k) + cost(k + 1, j) + d_i \cdot d_{k+1} \cdot d_{j+1}$$

  - Overlapping subproblems:

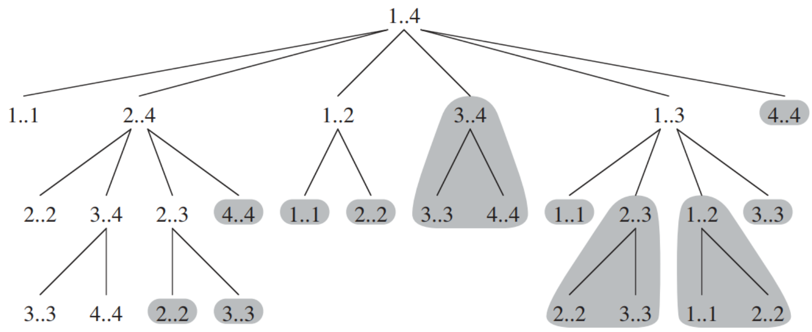


Figure 15.7 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Each node contains the parameters i and j. The computations performed in a shaded subtree are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN.

To memoize, we need a 2D-array. For order of filling, notes

1. Base cases: $cost(i, i)$ for all i.
2. $cost(i, j)$ needs $\{cost(i, i), \ldots, cost(i, j - 1)\}$ and $\{cost(i + 1, j), \ldots, cost(j, j)\}$.

i↓ j→	0	1	2	3	4
0					
1		(1, 1)	(1, 2)	(1, 3)	
2				(2, 3)	
3				(3, 3)	
4					

- Palindrome Partitioning. Similar to above. Reference.

3 Todos

First five problems from CSES.