Dynamic Programming Optimizations

Baraa Armoush

DP Optimizations

- The main objective of Dynamic Programming is to reduce the complexity of the program from exponential to polynomial.
- It do that dependent on the overlap of subproblems.
- But sometimes the naïve DP approach is not enough efficient.
- Top-Down version of DP can’t be improved because the values are calculated in a random order.
- Bottom-Up version of DP can be improved because we can calculate the values of DP in an order we specify.

DP Optimizations

- Memory Optimizations.
- Time Optimizations.

Memory Optimizations

- Rolling Table.
- Some Dimensions are Linearly connected.
- Instead of use Dp[i] as the value if we consider the i-th element and we try to pick or leave it, we use Dp[i] as the value if we pick this element and from it we iterative over the next element (ex: LIS).

Memory Optimizations

- Problems
  - Red-Green Towers

Time Optimizations

- Prefix Sum.
- Segment Tree.
- Set or Deque.
- Matrix.

Time Optimizations – Prefix Sum

- $Dp[i] = \sum_{j=L_i}^{j=R_i} Dp[j]$
- $1 \leq i \leq n\ , 0 \leq L_i \leq R_i < i$
- The previous formula has quadratic complexity.
- If we calculate the value of Dp for each element from 1 to n, we can compute in parallel Prefix Sum of the values of Dp.
- $Prefix[i] = Dp[0] + Dp[1] + Dp[2] + \cdots + Dp[i] = \sum_{j=0}^{j=i} Dp[j]$
- Then the Dp formula become :
- $Dp[i] = Prefix[R_i] - Prefix[L_i - 1]$
- Which has a linear complexity.

Time Optimizations – Prefix Sum

- $Dp[i][j] = \sum_{k=L_{ij}}^{k=R_{ij}} Dp[i-1][k]$
- The previous formula has cubic complexity.
- If we calculate the value of Dp for each row from 1 to n, when we need to compute the values of Dp in the i-th row, first we can build a prefix sum on the values of Dp in the (i – 1)-th row and the formula become :
- $Dp[i][j] = Prefix[i-1][R_{ij}] - Prefix[i-1][L_{ij}-1]$
- Which has a quadratic complexity.
- The same idea is correct if we have more than two dimensions.

Time Optimizations – Prefix Sum

- Problems
  - Candies
  - Riding in a Lift
  - New Year and Ancient Prophecy
  - Vasya and Array

Time Optimizations – Segment Tree

- $Dp[i] = \min_{L_i \le j \le R_i} Dp[j]$
- $1 \le i \le n\ , 0 \le L_i \le R_i < i$
- The previous formula has quadratic complexity.
- If we calculate the value of Dp for each element from 1 to n, we can compute in parallel Minimum Segment Tree of the values of Dp.
- Then the Dp formula become :
- $Dp[i] = SegmentTree.QueryMin(L_i, R_i)$
- Which has a complexity $O(n * \log(n))$.
- The same idea is correct if we have more dimensions.

Time Optimizations – Segment Tree

- Example
  - Longest Increasing Subsequence.
- Problems
  - Flowers
  - Babaei and Birthday Cake
  - Hanoi Factory

Time Optimizations – Lazy Segment Tree

- $Dp[i] = \min_{L_i \le j \le R_i} (Cost(i, j) + Dp[j])$
- $1 \le i \le n\ , 0 \le L_i \le R_i < i$
- Some formulas can be improved using Lazy Segment Tree (depended on the function *Cost*).
- Problems
  - The Bakery
  - Intervals
  - Animal Observation (hard version)

Time Optimizations – Set or Deque

- $Dp[i] = \min_{L_i \leq j \leq R_i} Dp[j]$
- $1 \leq i \leq n$, $0 \leq L_i \leq R_i < i$, $\boldsymbol{L_i \leq L_{i+1}}$, $\boldsymbol{R_i \leq R_{i+1}}$
- The previous formula has quadratic complexity.
- The ranges $[L_i, R_i]$ form a sliding window.
- We can use set or deque to know the minimum value in the window at any point of time.
- Set complexity : $O(n * \log(n))$
- Deque complexity : $O(n)$

Time Optimizations – Set or Deque

- Problems
  - Cashback
  - Strip
  - Pictures with Kittens
  - Bowling for Numbers++

Time Optimizations – Matrix

- $Dp[i] = \sum_{j=1}^{j=k} a_j * Dp[i-j]$
- $Dp[i] = a_1 * Dp[i-1] + a_2 * Dp[i-2] + \cdots + a_k * Dp[i-k]$
- $k \leq i \leq n$
- The previous formula has complexity $O(n * k)$

Time Optimizations – Matrix

- $k = 3$

$$[Dp_0 \quad Dp_1 \quad Dp_2] * \begin{bmatrix} 0 & 0 & a_3 \\ 1 & 0 & a_2 \\ 0 & 1 & a_1 \end{bmatrix}$$

$$= [Dp_1 \quad Dp_2 \quad (a_1 * Dp_2 + a_2 * Dp_1 + a_3 * Dp_0)]$$

$$= [Dp_1 \quad Dp_2 \quad Dp_3]$$

Time Optimizations – Matrix

- $k = 3$

$$[Dp_1 \quad Dp_2 \quad Dp_3] * \begin{bmatrix} 0 & 0 & a_3 \\ 1 & 0 & a_2 \\ 0 & 1 & a_1 \end{bmatrix}$$

$$= [Dp_2 \quad Dp_3 \quad (a_1 * Dp_3 + a_2 * Dp_2 + a_3 * Dp_1)]$$

$$= [Dp_2 \quad Dp_3 \quad Dp_4]$$

Time Optimizations – Matrix

- $k = 3$

$$[Dp_1 \quad Dp_2 \quad Dp_3] * \begin{bmatrix} 0 & 0 & a_3 \\ 1 & 0 & a_2 \\ 0 & 1 & a_1 \end{bmatrix}$$

$$= [Dp_0 \quad Dp_1 \quad Dp_2] * \begin{bmatrix} 0 & 0 & a_3 \\ 1 & 0 & a_2 \\ 0 & 1 & a_1 \end{bmatrix} * \begin{bmatrix} 0 & 0 & a_3 \\ 1 & 0 & a_2 \\ 0 & 1 & a_1 \end{bmatrix}$$

$$= [Dp_2 \quad Dp_3 \quad Dp_4]$$

Time Optimizations – Matrix

- $k = 3$

$$[Dp_0 \quad Dp_1 \quad Dp_2] * \begin{bmatrix} 0 & 0 & a_3 \\ 1 & 0 & a_2 \\ 0 & 1 & a_1 \end{bmatrix}^n$$

$$= [Dp_n \quad Dp_{n+1} \quad Dp_{n+2}]$$

Time Optimizations – Matrix

- In General

$$[Dp_0 \quad Dp_1 \quad \dots \quad Dp_{k-1}] * \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 & a_k \\ 1 & 0 & \cdots & 0 & 0 & a_{k-1} \\ 0 & 1 & \cdots & 0 & 0 & a_{k-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & a_2 \\ 0 & 0 & \cdots & 0 & 1 & a_1 \end{bmatrix}^n$$

$$= [Dp_n \quad Dp_{n+1} \quad \dots \quad Dp_{n+k-1}]$$

- Complexity $O(k^3 * \log(n))$

Time Optimizations – Matrix

- Problems
  - Magic Gems
  - Walk
  - RGB plants
  - Apple Trees
  - Xor-sequences
  - GukiZ and Binary Operations
  - PLEASE

- Problems
  - Darth Vader and Tree
  - Sonya and Informatics
  - Wet Shark and Blocks
  - Product Oriented Recurrence
  - Dexterina’s Lab
  - Okabe and El Psy Kongroo

To Be Continued