Dynamic Programming (I)

Chin Ka Wang {rina__owo}

2025-05-23

Content

1. Introduction to Dynamic Programming
2. Classic Linear DP Problem
    1. Maximum Subarray Sum (MSS)
    2. Longest Common Subsequence (LCS)
    3. Longest Increasing Subsequence (LIS)

DP is the art of breaking down a complex problem into a smaller subproblems.

Content

1. **Introduction to Dynamic Programming**
2. Classic Linear DP Problem
    1. Maximum Subarray Sum (MSS)
    2. Longest Common Subsequence (LCS)
    3. Longest Increasing Subsequence (LIS)

What is *Dynamic programming*?

Dynamic programming (DP) is a method for solving complex problems by breaking down the original problem into relatively simpler sub-problems.

Therefore, DP is often used to find the optimistic planning or decision in a complicated task. Sometimes, DP problems will mix with other data structures or optimization tricks.

The key of DP is to "memorize" something that have been calculated before, which is a trick of "**Trading Space for Time**".

Terminologies

- **Base Case(s)**
  - The simplest scenario(s) where the answer is immediately known.
- **State**
  - Store all essential information to define a subproblem.
- **Transitional Formula**
  - Defines the relationship between subproblems.

Terminologies

- **Optimal Substructure**
  - The optimal solution to the whole problem can be constructed from optimal solutions to its subproblems.
  - In other perspective, the optimal solution of the larger problem cannot depend on a non-optimal solution of the smaller problem.
  - DP works because of this property.

Terminologies

- **Overlapping Subproblems**
  - The problem can be broken down into smaller pieces that are reused multiple times.
  - Therefore, we will make use of memoization (not memorization) in DP.
- **Aftereffect**
  - The optimal solution of all subproblems should be determined and fixed.
- **Calculation Order**
  - To use DP effectively, calculate subproblem solutions in an order where you have the needed sub-solutions ready when you need them.
  - Often trivial.

Procedure to tackle a DP problem

1. Observe that the problem does not have aftereffect
2. Define the DP state (as well as the base case)
3. Figure out the relationship between states (sometimes adjacent)
4. Find out the transitional formula between states
5. Iterate from the base case to the result desired

Procedure to tackle a DP problem

We will use the problem Decode III as a demonstration of the procedure in tackling DP problem.

Detailed solution of the problem will not discuss here.

Simple problem restatement: find out the number of ways to insert whitespaces in a morse code message such that the message will be valid.

Procedure to tackle a DP problem

1. Observe that the problem does not have aftereffect

Assume we have split the message into two partition, *A* and *B*.

Obviously, there is no aftereffect for *A* no matter how many whitespaces inserted in *B*.

Procedure to tackle a DP problem

2. Define the DP state (as well as the base case)

Based on the idea, we can try to "split" the message in as many ways as possible.

Define the state $dp[i]$ as the number of ways to partition all previous *i* characters.

The base case $dp[0]$ should be 1.

Procedure to tackle a DP problem

3. Figure out the relationship between states (sometimes adjacent)

Note that we can only split the last partition with the length of [1, 5].

Therefore, we should try to obtain all possible partitions. When the length of the last partition is x, the number of ways to obtain the last partition is $dp[i-x]$.

Procedure to tackle a DP problem

4. Find out the transitional formula between states

Based on the idea before, we can obtain the transitional formula:

$$dp[i] = \sum_{k=i-5}^{i-1} \left(dp[k] \times \left[s[k+1 \,..\, i] \text{ is a valid partition} \right] \right)$$

Procedure to tackle a DP problem

5. Iterate from the base case to the result desired

We can iterate from $dp[0]$ to $dp[N]$ and the result is obtained.

The calculation order is trivial.

How to get better in DP?

- Most effective: Do more DP problems.
- Recite some common patterns in DP problems.
- Learn some DP optimization tricks.
- Draw the tabulation first before solving the DP problem.

Content

1. Introduction to Dynamic Programming
2. **Classic Linear DP Problem**
    1. Maximum Subarray Sum (MSS)
    2. Longest Common Subsequence (LCS)
    3. Longest Increasing Subsequence (LIS)

Classic Linear DP Problems

From now on, we will discuss some classic linear DP problems.

It is recommended that you fully understand all problems discussed and recite all deductions.

Content

1. Introduction to Dynamic Programming
2. Classic Linear DP Problem
    1. **Maximum Subarray Sum (MSS)**
    2. Longest Common Subsequence (LCS)
    3. Longest Increasing Subsequence (LIS)

Maximum Subarray Sum

Maximum Subarray Sum

- Given a sequence *a* with length *n*, find the maximum subarray sum of *a*.

Note: subarray is any **contiguous part of an array**. For example, an array [4, 5, -1] is a subarray of [1, 3, 4, 5, -1, 10]. However, [3, 5, 10] is not a subarray of [1, 3, 4, 5, -1, 10].

Maximum Subarray Sum

We can naïvely brute force the solution with nested for loops. Nevertheless, we can precompute the prefix sum of the array.

>= $O(n^2)$. **TLE.** ☹

Maximum Subarray Sum

Denote $a[i : j]$ be the sum of elements in the subarray of a between index i and j. Assume a is 1-based.

Trivially, $a[i : j + 1] = a[i : j] + a[j + 1]$.

Let say, for example, $a[k : j]$ is subarray that it has the maximum subarray sum of a that starts with index 1 and ends with index j.

How can we know the maximum subarray sum of a from index 1 to **j+1**?

Maximum Subarray Sum

Which of the options below may be the possible answer(s)?

- $a[k : j]$
- $a[k : j + 1]$
- $a[j + 1]$

Maximum Subarray Sum

Which of the options below may be the possible answer(s)?

- $a[k:j]$ ✅
- $a[k:j+1]$ ✅
- $a[j+1]$ ✅

Maximum Subarray Sum

Why are the options below impossible as the answer?

- $a[k + 1 : j + 1]$
- $a[k - 1 : j + 1]$
- $a[k : j - 1]$

Maximum Subarray Sum

Define $dp[i]$ as the maximum subarray sum of a where the subarray ends at index i.

- a[k : i + 1], a[i + 1] → ends with i + 1
- a[k : i] → ends with i

We do not need to care about a[k : i] when computing $dp[i + 1]$.

Maximum Subarray Sum

Therefore, we can obtain the transitional formula:

$$dp[i] = \max\left(dp[i-1] + a[i], a[i]\right)$$

Base case: $dp[0] = 0$

Time complexity: O(n) ☺

Content

1. Introduction to Dynamic Programming
2. Classic Linear DP Problem
    1. Maximum Subarray Sum (MSS)
    2. **Longest Common Subsequence (LCS)**
    3. Longest Increasing Subsequence (LIS)

Longest Common Subsequence

Longest Common Subsequence

- Given two sequence S and T consists of capital letters only. Find the length of the longest common subsequence of S and T.

Note that a subsequence is the sequence by deleting some elements (or no elements) in the original sequence without changing its order.

Longest Common Subsequence

Brute force solution: idk

I think brute force solution of these kinds of problems are even harder than achieving the DP solution.

Longest Common Subsequence

Let $dp[i][j]$ be the longest common subsequence of S[1 : i] and T[1 : j], assume both S and T are 1-based.

Note that there is no need to end with S[i] and T[j]!

Let's see what will happen if we consider S[1 : i + 1] and T[1 : j + 1].

Longest Common Subsequence

If we want to include *S*[i+1] in LCS but not *T*[j+1], then *dp*[i+1][j+1] can be *dp*[i+1][j].

If we want to include *T*[i+1] in LCS but not *S*[j+1], then *dp*[i+1][j+1] can be *dp*[i][j+1].

How about we want to include both?

- If S[i+1] != T[j+1], then obviously *dp*[i+1][j+1] = max(*dp*[i+1][j], *dp*[i][j+1]).

Longest Common Subsequence

How about S[i+1] = T[j+1]?

We can count them as an individual “pair” and append them.

Therefore, *dp*[i+1][j+1] can be *dp*[i][j] + 1.

Longest Common Subsequence

Therefore, we can obtain the transitional formula:

- For S[i] != T[j], $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$
- For S[i] = T[j], $dp[i][j] = \max(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]+1)$

Base cases: *dp*[*i*][0] = *dp*[0][*i*] = 0 for any *i*

Time Complexity: $\boldsymbol{O}(|\boldsymbol{S}| * |\boldsymbol{T}|)$

Longest Common Subsequence

This only gives us the length of the LCS.

If we want to know the actual subsequence, just record what we've chosen for each i and j, then **backtrack**!

Backtracking is a trick often used to retrieve the actual result in DP.

Content

1. Introduction to Dynamic Programming
2. Classic Linear DP Problem
   1. Maximum Subarray Sum (MSS)
   2. Longest Common Subsequence (LCS)
   3. **Longest Increasing Subsequence (LIS)**

Longest Increasing Subsequence

[Longest Increasing Subsequence]

- Given a sequence a with length n, find the length of the longest increasing subsequence of a.

Longest Increasing Subsequence

Brute Force Solution: idk

Longest Increasing Subsequence

Let *dp*[i] = the LIS of *a*[1 : i] that ended with *a*[i] (1-based).

Can we use a similar transitional formula in Maximum Subarray Sum?

No.

a[i] may not be greater than *a*[i - 1]. We need to consider all *dp*[j] where 1 <= j < i.

Longest Increasing Subsequence

By considering all cases, we can obtain the transitional formula.

$$dp[i] = \max_{0 \le j < i} \left(dp[j] \times \left[a[j] < a[i] \right] \right) + 1$$

Time complexity: $O(n^2)$, enough for AC.

Can we do better?

Longest Increasing Subsequence

Actually, a time of O(n) on searching for *a*[j] < *a*[i] is wasted!

How can we improve this?

Note that *a*[i] can only append to the subsequence that the last element of that is smaller than *a*[i]. If it is possible, the last element of the new subsequence become *a*[i].

Longest Increasing Subsequence

However, what the required last element actually is doesn’t really matter. We just care about if it is smaller than $a[i]$. Therefore, we can just find the **smallest last element**.

Why?

Longest Increasing Subsequence

arr/key	10	5	9	2	3	7	101	8
$f/value$	1	1	2	1	2	3	4	?

$arr/elem$	10, 5, 2	$<$9, 3	$<$ 7	$<$101
$f/index$	1	2	3	4

Longest Increasing Subsequence

If we store the corresponding “smallest last element” of all lengths of LIS. We can find that it is always increasing → Monotonicity!

If monotonicity exist, we can perform **binary search**!

Why is it always increasing?

- The proof is quite trivial so it is left as exercise. ☺

After optimizing, time complexity: O(nlogn)

Practice Problem

- Z0063 Maximum Subarray Sum
- Z0064 Longest Increasing Subsequence
- Z0065 Longest Common Subsequence

Q&A