

# Dynamic Programming (II)

Chin Ka Wang {rina\_\_owo}

2025-06-18

# Content

1. Knapsack DP
  - 0/1 Knapsack
  - Unbounded Knapsack (UKP)
  - Bounded Knapsack (BKP)
2. Interval DP
  - Stone Merging Problem
  - Longest Palindromic Subsequence (LPS)
  - Post Office Problem
3. Optimization
  - Rolling Array
  - Binary Grouping

# Content

1. **Knapsack DP**
  - 0/1 Knapsack
  - Unbounded Knapsack (UKP)
  - Bounded Knapsack (BKP)
2. Interval DP
  - Stone Merging Problem
  - Longest Palindromic Subsequence (LPS)
  - Post Office Problem
3. Optimization
  - Rolling Array
  - Binary Grouping

## Knapsack DP

The **Knapsack Problem** is a classic dynamic programming (DP) problem.

You are given a set of items, each with a weight and a value, and you need to maximize the total value without exceeding a given weight capacity of the knapsack.

# Content

1. Knapsack DP
  - 0/1 Knapsack
  - Unbounded Knapsack (UKP)
  - Bounded Knapsack (BKP)
2. Interval DP
  - Stone Merging Problem
  - Longest Palindromic Subsequence (LPS)
  - Post Office Problem
3. Optimization
  - Rolling Array
  - Binary Grouping

## 0/1 Knapsack

There are  $N$  items and a knapsack with capacity  $M$ .

The weight of the  $i^{th}$  item is  $w_i$ , the value is  $v_i$ .

Find out the largest total cost of the items that the knapsack can afford.

輸入	執行	輸出	完成 (0.001s)
4 6 1 4 2 6 3 12 2 7		23	

## 0/1 Knapsack

Let's observe what will happen before and after we put the  $i^{th}$  item into the knapsack:

Let  $v$  be the current value,  $c$  be the remaining capacity.

Before:  $(v, c)$

After:  $(v + v_i, c - w_i)$

## 0/1 Knapsack

Not considering the capacity, assume we have processed the previous  $i - 1$  items, there are only two situations:


( $m_i$  means the max value for the first  $i$  items)

Value of putting the  $i^{th}$  item =  $m_{i-1} + v_i$

Value of not putting the  $i^{th}$  item =  $m_{i-1}$

By combining two situations, max value the knapsack can carry for the first  $i$  item is:

$\max(m_{i-1} + v_i, m_{i-1})$





## 0/1 Knapsack

Note that no matter how we put the items, the total weight will always increase.



When we process the  $i^{th}$  item with capacity  $j$ , it never affect the max value of capacity  $< j$ . (No aftereffect)

## 0/1 Knapsack

- Let  $dp[i][j]$  be the max value of putting the first  $i$  items in a bag of capacity  $j$ .
- The answer of the question is  $DP[N][M]$ .

# Transitional Formula

Val	Wt	Item	Max Weight							
			0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0	1	1	4	5	5	5	5
5	4	3	0	1	1	4	5	6	6	9
7	5	4	0	1	1	4	5	7	8	9

# Transitional Formula

Transitional Formula: Link the relationship between the states of the first  $i - 1$  items and the first  $i$  item.

## Transitional Formula

$$dp[i][j] = \max(dp[i-1][j], dp[i-1][j-w_i] + v_i)$$

Base Cases:  $dp[0][k] = 0$  for any  $k$

Time Complexity:  $O(NM)$

Space Complexity:  $O(NM)$

## Calculation Order

Val	Wt	Item	Max Weight							
			0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0	1	1	4	5	5	5	5
5	4	3	0	1	1	4	5	6	6	9
7	5	4	0	1	1	4	5	7	8	9

stpc();

# Content

1. Knapsack DP
  - 0/1 Knapsack
  - Unbounded Knapsack (UKP)
  - Bounded Knapsack (BKP)
2. Interval DP
  - Stone Merging Problem
  - Longest Palindromic Subsequence (LPS)
  - Post Office Problem
3. Optimization
  - Rolling Array
  - Binary Grouping

# Unbounded Knapsack

UKP is a variation of 0/1 Knapsack Problem which allows unlimited repetition of items.



# Unbounded Knapsack

Naive solution:

For the  $i^{th}$  item, loop till the current weight exceed the capacity.

(Treat the  $i^{th}$  item as many different items)

$$f(i, j) = \max \begin{cases} f(i-1, j) \\ f(i-1, j - v[i] * 1) + w[i] * 1 \\ \dots \\ f(i-1, j - v[i] * k) + w[i] * k \quad k * v[i] \leq j \end{cases}$$

# Unbounded Knapsack

$$f(i, j) = \max \begin{cases} f(i-1, j) \\ f(i-1, j - v[i] * 1) + w[i] * 1 \\ \dots \\ f(i-1, j - v[i] * k) + w[i] * k & k * v[i] \leq j \end{cases}$$

Time Complexity:  $O(N^2M)$

Space Complexity:  $O(NM)$

# Unbounded Knapsack

$$f(i, j) = \max \begin{cases} f(i-1, j) \\ f(i-1, j - v[i] * 1) + w[i] * 1 \\ \dots \\ f(i-1, j - v[i] * k) + w[i] * k \quad k * v[i] \leq j \end{cases}$$

Note that the enumerate of  $j$  is from 0 to  $M \rightarrow$   
 we can get all info of  $j - k * v[i]$  before  $j \rightarrow$   
 $j - v[i]$  is already updated using the info of  $j - 2 * v[i] \rightarrow$   
 We just need to compare  $j$  and  $j - v[i]$

# Transitional Formula

Transitional Formula:

$$dp[i][j] = \max(dp[i - 1][j], dp[i][j - w_i] + v_i)$$

Base Cases:  $dp[0][k] = 0$  for any  $k$

Time Complexity:  $O(NM)$

## Review

Unbounded Knapsack:

$$dp[i][j] = \max(dp[i-1][j], dp[i][j - w_i] + v_i)$$

0-1 Knapsack:

$$dp[i][j] = \max(dp[i-1][j], dp[i-1][j - w_i] + v_i)$$

Very similar, don't mix them up.

# Content

1. **Knapsack DP**
  - 0/1 Knapsack
  - Unbounded Knapsack (UKP)
  - **Bounded Knapsack (BKP)**
2. Interval DP
  - Stone Merging Problem
  - Longest Palindromic Subsequence (LPS)
  - Post Office Problem
3. Optimization
  - Rolling Array
  - Binary Grouping

## Bounded Knapsack

BKP is also a variation of the 0/1 knapsack.

The difference from the 0/1 knapsack is that there are  $k_i$  items of each type instead of just one.

# Bounded Knapsack

Naïve Solution:

Treat it as a 0/1 Knapsack, where considering the  $k_i$  items of one type as  $k_i$  different items. Then use transitional formula of 0/1 knapsack to solve it.

Time Complexity:  $O(W \sum_{i=1}^N k_i)$

Space Complexity:  $O(NW)$



# Bounded Knapsack

Time Complexity can be optimized using monotonic queue or the binary grouping method.

We will talk about it later in section 3.

# Content

1. Knapsack DP
  - 0/1 Knapsack
  - Unbounded Knapsack (UKP)
  - Bounded Knapsack (BKP)
2. **Interval DP**
  - Stone Merging Problem
  - Longest Palindromic Subsequence (LPS)
  - Post Office Problem
3. Optimization
  - Rolling Array
  - Binary Grouping

## Interval DP

**Interval DP** is a dynamic programming technique used to solve problems.

The solution involves optimizing or counting over **subintervals** of a given sequence (e.g., arrays, strings). It breaks down the problem into smaller subproblems defined over intervals and combines their solutions.

## Interval DP

In simple words, **Interval DP** is often used in problems requiring you to merge (or split) subintervals while each operation worth a cost.

The target of this kind of problem is to calculate the max/min cost to achieve a target state of the input.

# Content

1. Knapsack DP
  - 0/1 Knapsack
  - Unbounded Knapsack (UKP)
  - Bounded Knapsack (BKP)
2. Interval DP
  - Stone Merging Problem
  - Longest Palindromic Subsequence (LPS)
  - Post Office Problem
3. Optimization
  - Rolling Array
  - Binary Grouping

# Stone Merging Problem

There are  $N$  stones arranged in a chain. The  $i^{th}$  stone is marked a score  $a_i$  on it.

In each operation, you merge two adjacent piles of stones into one, and the score gained is equal to the sum of the numbers on the stones in the merged pile. The goal is to maximize the total score.

# Stone Merging Problem

Knapsack DP no longer helps us.

We are not considering whether to put an item, but merging two subintervals.

The technique of **Interval DP** is needed in this case.

# Stone Merging Problem

Consider merging two stones  $a_1$  and  $a_2$ .



# Stone Merging Problem

Consider merging two stones  $a_1$  and  $a_2$ .

Easy.

Score we can gain =  $a_1 + a_2$ .

# Stone Merging Problem

Consider merging two stones  $a_1$  and  $a_2$ .

Easy.

Score we can gain =  $a_1 + a_2$ .

What about three or more?

## Stone Merging Problem

To get a result of 3 stones merged together, the last step must be merging two piles.

For example, to get the pile  $a_{1...3}$ ,  
we must merge  $a_1$  to  $a_{2...3}$  or merge  $a_3$  to  $a_{1...2}$ .

Trivially, it's impossible to merge  $a_2$  to some piles to get  $a_{1...3}$ .

## Stone Merging Problem

Therefore, to get the pile  $a_{1...3}$ , we can either have score:

1. Max score got from the pile  $a_{2...3} + \sum_{i=1}^3 a_i$
2. Max score got from the pile  $a_{1...2} + \sum_{i=1}^3 a_i$

Maximum score we can get from pile  $a_{1...3}$   
=  $\max(\text{Case 1, Case 2})$

## Stone Merging Problem

It's not hard to derive that, to get the pile  $a_{i...j}$ , we can have score:

Max score of  $a_{i...k} + a_{k+1...j} + \sum_{t=i}^j a_t$  where  $k \in [i, j - 1]$ .

Max score of  $a_{i...j} = \max(k = i, k = i + 1, \dots, k = j - 1)$ .

## Stone Merging Problem

- Let  $dp[i][j]$  be the max score got from the pile  $a_{i...j}$ .
- The answer of the question is  $dp[1][N]$ .

## Transitional Formula

$$dp[i][j] = \max_{k \in [i, j-1]} dp[i][k] + dp[k+1][j] + \sum_{t=i}^j a_t$$

Base Cases:  $dp[i][j] = 1e9$  for any  $i, j$ ; except  $dp[k][k] = 0$  for any  $k$

Time Complexity:  $O(N^3)$

Space Complexity:  $O(N^2)$

# Calculation Order

How can we enumerate i, j and k?

```
for (int i = 1; i <= N - 1; i++)  
    for (int j = i + 1; j <= N; j++)  
        for (int k = i; k <= j - 1; k++)  
            // Do something
```



## Calculation Order

How can we enumerate i, j and k?

```
for (int i = 1; i <= N - 1; i++)  
    for (int j = i + 1; j <= N; j++)  
        for (int k = i; k <= j - 1; k++)  
            // Do something
```



As we need to know all values of  $dp[i][k]$  and  $dp[k+1][j]$  when calculating  $dp[i][j]$ .

stpc();

## Calculation Order

Note that all  $dp[i][k]$  and  $dp[k+1][j]$  are subintervals of  $dp[i][j]$ .

We can introduce variable “**len**” (stands for length) to facilitate enumeration.

Thus, we can enumerate len from small to large (subinterval length) to ensure we got all info of smaller subintervals before computing a larger interval.

For each len, enumerate the starting index  $i$  and compute  $j = i + \text{len} - 1$ .

Enumerate the split point  $k$  between  $i$  and  $j$ , updating  $dp[i][j]$  by the transitional formula.

## Calculation Order

```
for (int len = 1; len <= N; len++)  
    for (int i = 1; i + len - 1 <= N; i++)  
        int j = i + len - 1;  
        for (int k = i; k <= j - 1; k++)  
            // Transitional Formula
```



stpc();

# Content

1. Knapsack DP
  - 0/1 Knapsack
  - Unbounded Knapsack (UKP)
  - Bounded Knapsack (BKP)
2. **Interval DP**
  - Stone Merging Problem
  - **Longest Palindromic Subsequence (LPS)**
  - Post Office Problem
3. Optimization
  - Rolling Array
  - Binary Grouping

# Longest Palindromic Subsequence

Given a sequence  $a$  with length  $n$ , find the length of the longest palindromic subsequence of  $a$ .

If a subsequence is **palindromic**, it's equal to itself after reverse.

# Longest Palindromic Subsequence

A solution of this problem is to convert it into a LCS Problem.

As we will mainly focus on Interval DP in this slide, the detail of the LCS solution is left as exercise and it should be easily figured out.

# Longest Palindromic Subsequence

Let's talk about the Interval DP solution.

The idea is actually pretty similar to LCS.

Let  $dp[i][j]$  be the longest palindromic subsequence of  $a[i : j]$ ,  
assume  $a$  is 1-based.

Note that there is no need to start with  $a[i]$  and ends with  $a[j]$ .

# Longest Palindromic Subsequence

Let's consider an interval  $a[i : j]$ .

Just like the LCS Problem,

If  $a_i \neq a_j$ ,  $dp[i][j]$  can be either transited from  $dp[i + 1][j]$  or  $dp[i][j - 1]$ .

If  $a_i = a_j$ ,  $dp[i][j]$  can be either transited from  $dp[i + 1][j]$ ,  $dp[i][j - 1]$  or  $dp[i + 1][j - 1]$ .

However, as larger interval are transited from subintervals, LPS is classified as Interval DP problem instead of Linear DP. Though, its idea is extremely similar to LCS.



## Transitional Formula

We can obtain the transitional formula easily:

- For  $a[i] \neq a[j]$ ,  $dp[i][j] = \max(dp[i+1][j], dp[i][j-1])$
- For  $a[i] = a[j]$ ,  $dp[i][j] = \max(dp[i+1][j], dp[i][j-1], dp[i+1][j-1] + 2)$

Base Cases:  $dp[k][k] = 1$  for any  $k$

Time Complexity:  $O(n^2)$

## Calculation Order

Same as stone merging problem, we have to first enumerate “len” to ensure we get all info of smaller subintervals when computing a bigger interval, then enumerate the starting index  $i$ .  $j$  can be easily calculated by  $i$  and  $len$ .

The only difference is we don't need to enumerate  $k$  this time.

# Content

1. Knapsack DP
  - 0/1 Knapsack
  - Unbounded Knapsack (UKP)
  - Bounded Knapsack (BKP)
2. Interval DP
  - Stone Merging Problem
  - Longest Palindromic Subsequence (LPS)
  - Post Office Problem
3. Optimization
  - Rolling Array
  - Binary Grouping

## Post Office Problem

There are  $n$  villages on some position on a number line.

$m$  post offices can be built in some villages.

Find the least sum of distances between the villages and the corresponding nearest post office of each village.

## Post Office Problem

Let's consider building only one post office in some villages.

Let  $w[i][j]$  be the minimum total distance of building a post office in the  $i^{th}$  village to the  $j^{th}$  village.

## Post Office Problem

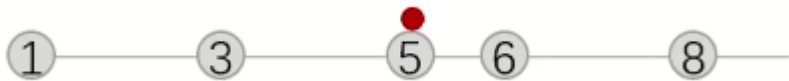
Let's consider building only one post office in some villages.

Let  $w[i][j]$  be the minimum total distance of building a post office in the  $i^{th}$  village to the  $j^{th}$  village.

Where should we build the post office to make the total distance the least?

## Post Office Problem

It can be easily observed and proven that the spot that we should build the post office is the **median** of those villages.



No. of villages is odd, build in median.



No. of villages is even, build in any of the two villages at the middle. (5 or 6)

## Post Office Problem

We can precompute all  $w[i][j]$  by recursion 遞推 (not 遞歸!)

(Although both methods are interchangeable but 遞推 is more readable so I recommend 遞推)

$$w[i][j] = w[i][j - 1] + (x[j] - x[\frac{i+j}{2}])$$

The distance between  $j^{th}$  village  
and the post office (array  $x$  stores x-coord)

This would be useful for our later works. You will know why soon.



## Post Office Problem

Now we already done half of the question.

Let's define  $dp[i][j]$  be minimum total distance of building  $j$  post offices in the first  $i^{th}$  village.

The answer of the question is  $dp[n][m]$ .

# Post Office Problem

How can we transit between states?

# Post Office Problem

How can we transit between states?

This question is very similar to **Decode III**.

Let's recall it.

# Post Office Problem

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

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

## Post Office Problem

The post office problem have to consider one more dimension, **the number of post office**.

You may observe that building a post office is actually **splitting** an interval. (a new office means some village will attach to it instead of attaching to original one)

**Interval DP** can be used!

## Post Office Problem

Consider all ways of “splitting” of the first  $i^{th}$  village:

$$v[1 : i] = v[1 : k] \cup v[k + 1 : i] \text{ where } k \in [1, i]$$

Assume there is already  $j - 1$  post offices built in  $v[1 : k]$ , and we have to build one village in  $v[k+1 : i]$ ,

$$dp[i][j] = dp[k][j - 1] + w[k + 1][i]$$

Note that we need  $O(n^2)$  time if we calculate  $w[k][i]$  every time but only  $O(n)$  if we precompute all of it by recursion.

## Transitional Formula

Therefore, transition formula would be:

$$dp[i][j] = \min_{k \in [0, i-1]} dp[k][j-1] + w[k+1][i]$$

Base Cases:  $dp[i][j] = 1e9$ ; except  $dp[1][k] = dp[k][0] = w[k][k] = 0$  for any  $k$

Time Complexity:  $O(n^2m)$

Space Complexity:  $O(nm)$

## Calculation Order

As this question uses “the first  $i^{th}$  village” to define the state, the calculation order is relatively straight forward.

We don’t need to introduce “len” this time as enumerating  $i$  ascendingly already ensured that smaller intervals are computed first.

Trivially, it should be  $j \rightarrow i \rightarrow k$ .



# Enrichment

Time complexity of this question can actually be optimized by some advanced technique called “**Quadrangle Optimization**” (or **Knuth Optimization**)!

This technique utilizes the monotonicity of some cost function (in this question is  $w[i][j]$ ) to optimize the time complexity of this question to  $O(nk)$ .

We can even use more advanced **WQS Binary Search** to solve the question in  $O(n)$ .

# Content

1. Knapsack DP
  - 0/1 Knapsack
  - Unbounded Knapsack (UKP)
  - Bounded Knapsack (BKP)
2. Interval DP
  - Stone Merging Problem
  - Longest Palindromic Subsequence (LPS)
  - Post Office Problem
3. Optimization
  - Rolling Array
  - Binary Grouping

# Optimization

Optimization is a common trick in DP questions.

Purpose of optimization is lower the time and space complexity of your algorithm to prevent TLE and MLE.

# Optimization

Common DP optimization tricks includes:

- Space Optimization:
  - Rolling Array
  - State Compression
- Time Optimization:
  - Pruning (Used in Decode III)
  - Monotonic Queue/Stack Optimization
  - Slope Optimization
  - Quadrangle Optimization
  - CDQ Divide and Conquer
  - WQS Binary Search

# Content

1. Knapsack DP
  - 0/1 Knapsack
  - Unbounded Knapsack (UKP)
  - Bounded Knapsack (BKP)
2. Interval DP
  - Stone Merging Problem
  - Longest Palindromic Subsequence (LPS)
  - Post Office Problem
3. Optimization
  - Rolling Array
  - Binary Grouping

## Rolling Array

**Rolling Array** is a space optimization technique that reduces the memory usage of dynamic programming by reusing or overwriting parts of the DP table that are no longer needed.

## Rolling Array

**Rolling Array** is a space optimization technique that reduces the memory usage of dynamic programming by reusing or overwriting parts of the DP table that are no longer needed.

Let's recall the transitional formula of 0/1 Knapsack.

## Rolling Array

**Rolling Array** is a space optimization technique that reduces the memory usage of dynamic programming by reusing or overwriting parts of the DP table that are no longer needed.

Let's recall the transitional formula of 0/1 Knapsack.

$$dp[i][j] = \max(dp[i-1][j], dp[i-1][j - w_i] + v_i)$$



## Rolling Array

$$dp[i][j] = \max(dp[i-1][j], dp[i-1][j-w_i] + v_i)$$

We can easily observe that we just need all info of  $i-1$  to get all info of  $i$ . That means the  $i-2, i-3, \dots, 0$  row of the DP table is no longer useful anymore and they are wasting our space.

Trivially, we can only use a DP table of only two rows, where the first row always stores the states of previous items, and the second row is used for calculating new states after considering the new item.

After it, copy all the element in the second row to the first row, consider a new item, and repeat this procedure.

## Rolling Array

However, this can be further optimized to only one row.

$$dp[j] = \max(d[j], dp[j - w_i] + v_i)$$

Can we directly overwrite the new info in one row with the same calculation order?

## Rolling Array

From the transitional formula,

$$dp[j] = \max(dp[j], dp[j - w_i] + v_i)$$

We can easily observe that we only need info of  $j - w_i$  to get info of  $j$ , which  $j - w_i < j$ . That means, all info larger than  $j$  is no longer useful when computing  $j$ . Therefore, we can overwrite them rather than overwriting info smaller than  $j$ .

In simple words, we need to reverse the order of enumeration of  $j$ , i.e. in descending order.

# Content

1. Knapsack DP
  - 0/1 Knapsack
  - Unbounded Knapsack (UKP)
  - Bounded Knapsack (BKP)
2. Interval DP
  - Stone Merging Problem
  - Longest Palindromic Subsequence (LPS)
  - Post Office Problem
3. Optimization
  - Rolling Array
  - Binary Grouping

## Binary Grouping

This optimization can be used in many scenarios, not only DP.

To understand this method, we have to first understand a fact:

**Any number can be represented by sum of  $2^k$  s.**

E.g.  $11 = 2^0 + 2^1 + 2^3$


Think about the binary representation of a number will easily understand this statement.

# Binary Grouping

Binary Grouping can be used to optimize the BKP, i.e. 0/1 Knapsack but there are  $k_i$  of each type of item.

In our previous solution without optimization, we need to do this:

$w_i$	$v_i$	No.
1	1	600



$w_i$	$v_i$	No.
1	1	1
1	1	1
1	1	1
1	1	1
⋮	⋮	⋮
1	1	1
1	1	1
1	1	1

Which did too many loops

# Binary Grouping

As  $600 = 1 + 2 + 4 + \dots + 128 + 256 + 89$ ,  
we can actually split the items like this:

$w_i$	$v_i$	No.
1	1	600



$w_i$	$v_i$	No.
1	1	1
2	2	1
4	4	1
8	8	1
$\vdots$	$\vdots$	$\vdots$
128	128	1
256	256	1
89	89	1

## Binary Grouping

Why group like  $600 = 1 + 2 + 4 + \dots + 128 + 256 + 89$ ,  
but not  $600 = 512 + 64 + 16 + 8$ , the binary representation of 600?



## Binary Grouping

Why group like  $600 = 1 + 2 + 4 + \dots + 128 + 256 + 89$ ,  
but not  $600 = 512 + 64 + 16 + 8$ , the binary representation of 600?

**Think clear about our goal!**

## Binary Grouping

Considering the split table, you can find that all cases can be represented by those numbers:

1. Putting 0 to 511 of the items can be represented by combinations of putting items weighted 1, 2, 4, ..., 256.
2. Putting 512 to 600 of the items can be represented by putting item weighted 89 + putting 423 to 511 of the items (which is same as case 1)

## Binary Grouping

Splitting 600 to 512, 64, 16, 8 is meaningless. It cannot ensure we considered all possible ways of “putting item”.

In other words, by just doing 0/1 Knapsack on the binary grouping table, we can obtain the optimal solution among all the possible ways of putting 600 same type of items.

## Binary Grouping

If you have the sense, you should know that “binary” related optimizations usually optimizes  $O(n)$  scale problem to  $O(\log n)$  scale.

Therefore, binary grouping optimization optimizes BKP from  $O(W \sum_{i=1}^N k_i)$  to  $O(W \sum_{i=1}^N \log k_i)$ .

## Q&A

---