Dynamic programming 1

TAI, WEI HSUAN

September 8, 2025

Outline

- Basic concept of dynamic programming
- How to conduct dynamic programming
- Longest Increasing Subsequence (LIS) problem
- Backpack problems

Basic concept of dynamic programming

Sometimes we need to solve a problem that can be divided into subproblems, and the solution to the original problem can be obtained by combining the solutions of the subproblems. This is the basic concept of dynamic programming: we ”record” the subsolutions of the problem, and then use these subsolutions to solve the original problem.

A simple example

Can you tell me the nth Fibonacci number?
You can't directly tell me the answer unless you use the mathematical formula(this is hard to calculate), but if you know the n-2th and n-1th Fibonacci numbers, you can easily calculate the nth Fibonacci number. This is the basic idea of dynamic programming: we can use the solutions of subproblems to solve the original problem.

Code for Fibonacci Numbers

Following is a simple code to calculate the nth Fibonacci number using recursion.

```
1        int fib(int n){
2            if (n<=1)return n;
3            return fib(n-1)+fib(n-2);
4        }
```

The time complexity of this code is $O(\phi^n)$ because we repeatly calculate the same Fibonacci numbers, which is inefficient.
To record the subsolutions, we can use an array to store the Fibonacci numbers we have calculated. This is the main concept of dynamic programming.

Code for Fibonacci Numbers with Dynamic Programming

```
1        int F[10005];
2        int fib(int n){
3            if(F[n])return F[n];
4            if (n<=1)return F[n]=n;
5            return F[n]=fib(n-1)+fib(n-2);
6        }
```

In the code above, we use an array F to store the Fibonacci numbers we have calculated. When we need to calculate the nth Fibonacci number, we first check if it is already in the array.
This way, we can reduce the time complexity to $O(n)$, which is much more efficient than the previous code.

Code for Fibonacci Numbers with Dynamic Programming

We can simplify the code further by using an iterative approach.

```
1         int F[10005];
2         F[0]=0, F[1]=1;
3         for(int i=2;i<=n;i++){
4             F[i]=F[i-1]+F[i-2];
5         }
```

A harder example: Vacation

Taro's summer vacation starts tomorrow, and he has decided to make plans for it now.
The vacation consists of N days. For each i ($1 \leq i \leq N$), Taro will choose one of the following activities and do it on the i-th day:

- A: Swim in the sea. Gain a_i points of happiness.
- B: Catch bugs in the mountains. Gain b_i points of happiness.
- C: Do homework at home. Gain c_i points of happiness.

As Taro gets bored easily, he cannot do the same activities for two or more consecutive days.
Find the maximum possible total points of happiness that Taro gains.

Analyze this problem

You can't obtain the answer just by intuitively thinking about it. Let's analyze the problem step by step.
Consider the first day, Taro can choose any of the three activities. For the second day, Taro have three sets of choices:

First Day Activity	**Second Day Options**
A (Swim)	B (Catch bugs) or C (Homework)
B (Catch bugs)	A (Swim) or C (Homework)
C (Homework)	A (Swim) or B (Catch bugs)

Analyze this problem

It is obvous that Taro would choose the highest happiness option for the second day.
Therefore, the maximum happiness for the second day can be calculated as follows:

$$max(A1 + max(B2, C2), B1 + max(A2, C2), C1 + max(A2, B2))$$

If we record the maximum happiness for each day and each activity, we can use the following formula to calculate the maximum happiness for each day:

Extend to more days

Now we have the maximum happiness for the first two days.With this concept, we can extend it to more days.
Just a little modification to the formula, define $dp[i][j]$ as the maximum happiness for the first i days, where j is the activity on the i-th day:

$$dp[i][0] = a_i + \max(dp[i-1][1], dp[i-1][2])$$

$$dp[i][1] = b_i + \max(dp[i-1][0], dp[i-1][2])$$

$$dp[i][2] = c_i + \max(dp[i-1][0], dp[i-1][1])$$

We can easily use a loop to calculate the maximum happiness for each day.

How to conduct dynamic programming

Here are the steps to conduct dynamic programming:

1. **Define the state**: Define what is ”dp[i][j]” in the problem.
2. **Define the transition**: Define the relationship between the states, which is the formula we derived above.
3. **Define the base case**: Remember to set the base case, it should cover all situations that won’t be reached by the transition.

LIS problem

The Longest Increasing Subsequence (LIS) problem is a classic problem in computer science. Given an array of integers, the goal is to find the length of the longest subsequence that is strictly increasing.
For example, given the array [10, 9, 2, 5, 3, 7, 101, 18], the longest increasing subsequence is [2, 3, 7, 101], and its length is 4.

Idea of Force Brute

The brute force solution is to generate all possible subsequences of the array and check if they are increasing. However, this approach has an exponential time complexity of $O(2^n)$, which is not feasible for large arrays.

Idea of Dynamic Programming

To execute the dynamic programming, we can define a state $dp[i]$ as the length of the longest increasing subsequence ending at index i.
How to find $dp[i]$? You just search the previous elements and find a number that is smaller than the current number, and then add 1 to the length of the longest increasing subsequence ending at that index.
The transition formula can be written as:

$$dp[i] = \max(dp[j] + 1) \text{ for all } j < i \text{ and } arr[j] < arr[i]$$

with a simple loop we can solve it. What is the time complexity?

Idea of Dynamic Programming

The time complexity of the above solution is $O(n^2)$, which is much better than the brute force solution.
Last step, we need to define the base case.
In this case, we do not simply define $d[0]$ or $dp[1]$, instead, for all i, we can set $dp[i] = 0$, so if there's no elements before i is smaller than $arr[i]$, the longest increasing subsequence ending at i is just itself.
On the other aspect, before we conduct the transition, every subsequences contains no elements.

Optimization

The $O(n^2)$ solution is still not efficient enough for large arrays, we can optimize the solution to $O(n \log n)$ by using binary search. We can maintain an array *tails* where $tails[i]$ is the minimum ending value of all increasing subsequences of length $i + 1$.

Backpack problems

The backpack problem is another classic problem in computer science.
You have a backpack with a maximum weight capacity, and you have a set of items, each with a weight and a value. The goal is to maximize the total value of the items you can put in the backpack without exceeding the weight capacity.
Here are some variations of the backpack problem:

- 0/1 Backpack Problem: You can either take an item or leave it.
- Fractional Backpack Problem: You can take a fraction of an item.
- Infinity Backpack Problem: You can take an item multiple times.
- etc...

01 Backpack Problem - State Definition

For each item, you can either take it or leave it. We can define a state **dp[i][j] as the maximum value of the backpack with a weight capacity of j and considering the first i items.**
With this state definition, you just output dp[N][W] where N is the number of items and W is the maximum weight capacity of the backpack.

01 Backpack Problem - Transition

For ith item, it have the weight w_i and value v_i.
Here are the two choices we can make:

- Do not take the item: $dp[i][j] = dp[i-1][j]$
- Take the item: $dp[i][j] = dp[i-1][j-w_i] + v_i$ (if $j \geq w_i$)

You may wonder why, when we take an item, we subtract its weight from the remaining capacity.
This is because we are evaluating the **capacity left** after including the item. In the DP table, we iterate over all items and all possible capacities, thus covering all possible combinations.

01 Backpack Problem - Base Case

The base case should cover the situations that cannot be reached by the transition. Consider the transition we defined above, we can see that if we have no items or the backpack has no capacity, the maximum value is zero.

- $dp[0][j] = 0$ for all $j \geq 0$ (no items)
- $dp[i][0] = 0$ for all $i \geq 0$ (zero capacity)

This means that if there are no items to choose from or the backpack has no capacity, the maximum value is zero.

Complexity Analysis

What is the time complexity and the space complexity of the 0/1 backpack problem?
We create a 2D array of size $N \times W$, where N is the number of items and W is the maximum weight capacity of the backpack.
The time complexity is $O(N \times W)$ because we iterate through all items and all capacities to fill the DP table. The space complexity is also $O(N \times W)$ due to the 2D array.

Optimize the Space Complexity

We observe that the current space complexity may cause memory issues for large inputs. If $N \times W$ is close to 10^9, it is impossible to store the DP table in memory.
To address this, we can optimize the space complexity to $O(W)$ by using a 1D array instead of a 2D array. Consider the transition we defined, we can find that the current state only depends on the previous state, which means we do not need to store the entire DP table.
We can use a single array to store the maximum value for each capacity, and update it iteratively.

Rolling Array Optimization

Consider that we just need to store the previous state, we can obtain the following formula:

$$dp[w][j\%2] = max(dp[w][(j-1)\%2], dp[w-w_i][(j-1)\%2] + v_i)$$

In this way, we only need to store the current and previous states, which reduces the space complexity to $O(W)$. The time complexity remains $O(N \times W)$, but the space complexity is now $O(W)$.

Still too Large?(Optional)

Observe the transition again, we can see that the current state is based on the previous state, this is why we need to store the previous state.
However, what if we update the DP table in reverse order? If we update the DP table from W to w_i, we can avoid overwriting the previous state, and we can still use a 1D array to store the maximum value for each capacity.
The transition can be written as:

$$dp[j] = \max(dp[j], dp[j - w_i] + v_i) \text{ for all } j \text{ from } W \text{ to } w_i$$

This way, we can still use a single array to store the maximum value for each capacity, and update it iteratively in reverse order. The time complexity remains $O(N \times W)$, and the space complexity is still $O(W)$.
This skill is optional because it seldom appears in the exam.

Infinity Backpack Problem

Similiar to 01 backpack problem, but you can take an item multiple times.
How to handle this problem?

Thoughts

Different from the 01 backpack problem, we can take an item multiple times. This means, if we do choose the ith item, we do not need to consider the previous items, we can just take the ith item again.
Therefore, the transition can be written as:

$$dp[i][j] = \max(dp[i-1][j], dp[i][j-w_i] + v_i) \text{ for all } j \geq w_i$$

Reference

- https://github.com/tianyicui/pack/blob/master/V2.pdf
- https://oi-wiki.org/dp/
- https://atcoder.jp/contests/dp/tasks
- https://leetcode.com/problem-list/dynamic-programming/