DP Knowledge

What is Dynamic Programming?

Dynamic programming is a method for solving problems by breaking them down into smaller subproblems, solving each one just once, and storing the results so we don't repeat the same work over and over again.

When to use DP?

It’s usually used when:

1. The problem can be divided into smaller **overlapping subproblems**
2. The solution to the whole problem depends on the solutions to those smaller parts — this is called **optimal substructure**

Overlapping Subproblems

The same subproblem will be repeatedly calculated between different subproblems.

> *Take the house robber problem as an example. If we try to list all possible robbing paths:*

> ```
> Path A: rob house 0 → skip 1 → rob 2 → skip 3 …
> Path B: skip house 0 → rob 1 → skip 2 → rob 3 …
> ```
>
> *In both paths, we might eventually ask: "What's the maximum money I can rob starting from house 2?"*
>
> *That subproblem — "starting from house 2" — shows up in multiple paths. That's what overlapping subproblems means: We're recalculating the same things many times. So instead of solving it repeatedly, we can just solve it once and store the result, which is exactly what dynamic programming does.*

Optimal Substructure

The optimal solution to the original problem can be derived from the optimal solutions to its subproblems.

> *"I can build the solution to the whole big problem by solving small problems."*
>
> *For example, I have to walk up 10 steps in a staircase, 1 or 2 steps at a time; if I know the maximum number of ways to reach the 8th step, and the maximum number of ways to reach the 9th step; then I can calculate the number of ways to reach the 10th step. This is called the optimal substructure.*

Basic Steps

1. **Determine the subproblem** Decide what `dp[i]` actually means.
2. **Define state transition equation** Figure out how to build `dp[i]` based on previous states like `dp[i - 1]`, `dp[i - 2]`, etc. This equation expresses the logic of "if I'm at step `i`, what options do I have?"

3. **Initialization** Specify the starting state, such as `dp[0] = 1`, `dp[1] = nums[0]`.
4. **Fill out the the DP table** Use a loop (usually `for i in range(...)`) to compute the result for every state from the bottom up.
5. **Return result** This might be `dp[n - 1]`, `dp[n]`, or sometimes the max/min among all `dp[i]` depending on the problem.

Types

1. **Linear DP** Problems where the solution at position `i` depends on previous positions in a 1D sequence.
2. **Knapsack DP** Problems involving combinations of items and capacities — usually maximizing or minimizing some value.
3. **Interval DP** Problems where the subproblems are defined on intervals or ranges within an array.
4. **Bitmask (State Compression) DP** When there are multiple dimensions of states, we compress them using bitmasks for efficiency.
5. **Tree DP** DP on trees, where each node's result depends on its children's subproblems.
6. **Memoization (Top-Down DP)** Recursive approach where results of subproblems are cached to avoid repeated computation.

Linear DP Example Analysis

---

198. House Robber

Firstly, we should **read the question and abstract key information**. Let's start by understanding the question.

We can extract four key pieces of information:

- Each house has some money inside.
- The only restriction is that we can't rob two adjacent houses, otherwise the alarm will go off.
- We are given an integer array nums, where each element is the amount of money in a house.
- Our goal is to return the maximum amount of money we can rob without alerting the police.

```
# house has money
# break into 2 adjacent houses will alert
# input: array nums -> nums[i] money of house
# output: max money in total without alert
```

Secondly, **try small example and observe behavior pattern**: find the most common method.

The most instinctive way is to list all the valid combinations that follow the rule: no two adjacent houses can be robbed. Let's try a few:

- Rob house 0 and house 2 → 1 + 3 = 4
- Rob house 1 → 2
- Rob house 1 and house 3 → invalid, they are adjacent
- Rob house 2 → 3

Among these, the best result is 4, which is the correct answer.

Now let's look at a slightly larger input: [2, 7, 9, 3, 1]. We try:

- Rob house 0, 2, and 4 → 2 + 9 + 1 = 12
- Rob house 1 and 3 → 7 + 3 = 10
- Rob house 1 and 4 → 7 + 1 = 8
- Rob house 2 and 4 → 9 + 1 = 10 (note: 2 and 4 are not adjacent)
- Rob house 0 and 3 → 2 + 3 = 5

The maximum total is 12, so that's our correct answer.

But clearly, as the number of houses grows, the number of valid combinations increases rapidly. It quickly becomes infeasible to list them all. Also, in some cases, we might skip two houses in a row, so the valid paths are not evenly spaced or easy to pattern-match. This makes brute-force not just slow, but hard to generalize.

So we need a more efficient way to model the problem. Now looking closer: For each house, if we rob it, we can't rob its adjacent neighbor.

Then, we begin to **recognize decision pattern and identify DP nature**: For every house, we're essentially facing the same decision: "steal" vs. "don't steal". That's a binary decision. And since we're not allowed to rob two adjacent houses, if we rob house `i`, we have to skip house `i - 1`. Let's try to express this relationship using values.

```
 # If we stole house i (i is the index of the number in array)
money[i] = money[i-2] + nums[i]

# if we not stole house i
money[i] = money[i-1]
```

Since we want the maximum amount of money, we take the better of the two:

```
money[i] = max(money[i-2] + nums[i], money[i-1])
```

All valid paths are simply combinations of these two choices. This creates a **uniform behavior pattern**: every house follows the same logic — rob or skip. And the best decision at position `i` depends only on previous results. That's what we call **optimal substructure** — we can build the big problem's solution from the solutions to smaller subproblems.

And since many different decision paths might reuse the result of the same house — like house `i - 2` or `i - 1` — we clearly have **overlapping subproblems** as well. That means we can use recursion + memoization, or bottom-up tabulation, to avoid redundant calculations.

Now that both properties of dynamic programming are satisfied, we know for sure: **this is a DP problem**.

Next step is to **build the DP process**. Let's actually build the DP solution.

We have decided:

- Subproblem: For each house, we want to decide: rob it or not, and we want the decision that gives us the most money.
- State transfer equation: The transition equation we came up with earlier was `dp[i] = max(dp[i - 1], dp[i - 2] + nums[i])`

In 5 dp basic steps, we have 3 steps to do:

- Define dp state:
    - The state we define (dp[i]) must be a solution to "answer we ultimately want". Let `dp[i]` represent the maximum amount of money we can rob from the first `i + 1` houses.
    - Its length should match **the full range of subproblems we need to solve**. So the length of the `dp` array needs to be exactly `len(nums)`.

- Initialization: we need to handle the initial values manually. For those subproblems whose values can be obtained directly or the starting states that cannot be deduced from the state transition equations, the initial values must be set manually. The state transition formula is `dp[i] = max(dp[i-1], dp[i-2] + nums[i])` . So `dp[i-1]` and `dp[i-2]` must exist， we must set the initial value.
- Build solution and fill out the rest of the DP table.

```
dp = [0] * len(nums)

dp[0] = nums[0]
dp[1] = max(nums[0], nums[1])

for i in range(2, len(nums)):
    dp[i] = max(dp[i-2] + nums[i], dp[i-1])

return dp[-1]
```

Now before we finish and run, we should also **think about edge cases**.

Edge cases are inputs that sit at the extremes — very small, very large, very empty, very full, and so on. They often cause bugs if we don't plan for them.

Let's look at the constraints from the problem:

- The number of houses: `1 <= nums.length <= 100`
- The money in each house: `0 <= nums[i] <= 400`

So here are a few questions we should ask ourselves — I like to call these the "edge case checklist":

- What's the minimum input size? （如 `nums.length = 1`）
- What's the maximum size? （如 `nums.length = 100`）
- Is it possible for each element to be 0, a negative number, or an extreme value?

- From which index can the DP formula actually start? If our initialization values are right?
- Are we at risk of illegal index access like `dp[-1]` or `dp[-2]`?

We must check those before applying the formula. Once we cover these edge cases, our solution will be solid and reliable for any valid input.

```
class Solution(object):
    def rob(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        # think about edge cases
        if len(nums) == 1:
            return nums[0]

        dp = [0] * len(nums)

        dp[0] = nums[0]
        dp[1] = max(nums[0], nums[1])

        for i in range(2, len(nums)):
            dp[i] = max(dp[i-2] + nums[i], dp[i-1])

        return dp[-1]
```

53. Maximum Subarray

Firstly, we should **read the question and abstract key information**. Let's start by understanding the question.

We can extract 3 key pieces of information:

- We're given an array of integers called `nums`,
- We're supposed to **find the maximum sum of a subarray**.
- A subarray means the elements have to be **contiguous** — no skipping around — and it can't be empty.

```
# input: array -> nums[integer]
# output: integer -> maximum sum of subarray
# subarray: continuous, no-empty
```

Secondly, let's **try a small example** and see if we can **spot a pattern**.

The most instinctive way to solve this is brute force — just try every possible subarray.

That means:

- For every possible starting point `i`,
- we try every possible ending point `j` (where `j >= i`),
- and for each subarray `nums[i...j]`, we calculate its sum.

Here's what the pseudocode looks like in plain logic:

```
max_sum = -infinity

for i from 0 to n-1:
    for j from i to n-1:
        subarray_sum = 0
        for k from i to j:
            subarray_sum += nums[k]
        max_sum = max(max_sum, subarray_sum)
```

So you've got **three nested loops**: The outer two to define every subarray, and the inner one to compute the sum each time. this approach is really slow — that's `O(n^3)` time. Even if we optimize the sum calculation, it's still `O(n^2)`, and for big arrays, that's way too slow.

So clearly, brute-force isn't going to cut it. We need a better plan — something faster and smarter.

Now we start thinking: Is there any way to avoid doing all that repeated work? Can we somehow make smarter choices and cut down unnecessary steps?

So instead of checking every single subarray, we ask ourselves: **At each step, can we make a decision — either to keep adding to the current subarray or to start a new one — based on what gives us a better result?**

Because we need to keep the sum of the current subarray as large as possible, the criterion is the sum of the subarrays. Which gives us a bigger sum?

We have two options:

```
# add
sum_subarray[i] = sum_subarray[i-1] + nums[i]

# not add and start new subarray
sum_subarray[i] = num[i]
```

So the final decision is:

```
sum_subarray[i] = max(sum_subarray[i-1] + nums[i], num[i])
```

In other words, if adding the current number to the previous sum gives us a better total, we keep going. If not, we cut off the old subarray and start fresh from this number.

This gives us a **uniform behavior pattern** — for every index, we're making the same kind of decision: **"Extend the previous subarray, or start a new one?"**

Once we see this kind of consistent local decision, and we realize that each decision only depends on the result from the previous step, we know this problem has:

- **Optimal substructure** — the best answer at i depends on the best answer at i-1.
- **Overlapping subproblems** — we reuse previously computed results.

This is a perfect fit for dynamic programming.

Next step is to **build the DP process**. Let's actually build the DP solution.

We have decided:

- Subproblem: Do I add the current number to the previous subarray, or do I start a new subarray from here to make the sum of the subarray as big as possible.
- State transfer equation: The transition equation we came up with earlier was `dp[i] = max(dp[i - 1] + nums[i], nums[i])`

In 5 dp basic steps, we have 3 steps to do:

- Define dp state:
    - The `dp[i]` represent **the maximum sum of a subarray that ends exactly at index i**. It must include `nums[i]`, but can either extend the previous subarray, or be a fresh start at this number.
    - Its length should match **the full range of subproblems we need to solve**. So the length of the `dp` array needs to be exactly `len(nums)`.
- Initialization: The state transition formula is `dp[i] = max(dp[i - 1] + nums[i], nums[i])` . We cannot get dp[0], because dp[0] is the first number in dp array. So `dp[0]` must be set manually.
- Build solution and fill out the rest of the DP table.

```
dp = [float("-inf")] * (len(nums))

dp[0] = nums[0]

for i in range(1, len(nums)):
  dp[i] = max(dp[i-1] + nums[i], nums[i])

return max(dp)
```

```
class Solution(object):
    def maxSubArray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        # input: array -> nums[integer]
        # output: integer -> maximum sum of subarray
        # subarray: continuous, no-empty
        dp = [float("-inf")] * (len(nums))

        dp[0] = nums[0]

        for i in range(1, len(nums)):
            dp[i] = max(dp[i-1] + nums[i], nums[i])

        return max(dp)
```

Now we should **think about edge cases**.

Let's look at the constraints from the problem:

- `1 <= nums.length <= 10^5` — so the input is guaranteed to be non-empty, we don't need to handle `nums = []`.

- `-10^4 <= nums[i] <= 10^4` — Each number can be as low as `-10^4`, so we can definitely have **all-negative arrays** like `[-2, -1, -3]`.

So, the worst-case edge case would be an array made up entirely of negative numbers. And we should make sure our code still returns the correct result — that is, the **least negative number**. Also, we should be careful with our **initial value** for max sum. If we accidentally initialize it as `0`, we might wrongly return 0 for an all-negative array. That's why we use:

```
dp[0] = nums[0]
```

This guarantees we're starting from the actual data, not some arbitrary default like 0.

And one last thing before we wrap up. We can actually **optimize the space** in our solution. Right now we're using a full `dp` array to store every step, but if you look at our state transition equation:

```
dp[i] = max(dp[i - 1] + nums[i], nums[i])
```

We're only ever using `dp[i - 1]` to calculate `dp[i]`. That means we don't really need to keep the whole array — just **the previous value**.

So we can replace the array with two variables:

- One to track the current subarray sum, and
- One to track the maximum sum we've seen so far.

Here's how it looks:

```
class Solution(object):
    def maxSubArray(self, nums):
        """
        :type nums: List[int]
```

```
        :rtype: int
        """
        # input: array -> nums[integer]
        # output: integer -> maximum sum of subarray
        # subarray: continuous, no-empty
        cur_sum = nums[0]
        maximum = nums[0]

        for i in range(1, len(nums)):
            cur_sum = max(cur_sum + nums[i], nums[i])
            maximum = max(maximum, cur_sum)

        return maximum
```

This gives us the same result, but with only O(1) space instead of O(n), and still O(n) time. This kind of optimization is super common in dynamic programming — once you understand the pattern and know you only need the last result, you can often reduce memory usage like this.