

Introduction to DP

Sept 14, 2024

Note: In many DP problems, the numbers get so large that they stop fitting in 32-bit and even 64-bit integers. To get around this, many sites will tell you to compute the answer modulo some prime $p \approx 10^9$. However, to simplify the presentation, I will pretend no overflow occurs.

Here is the list of problems. The solutions will start on the next page.

Problem 1 (Source: <https://leetcode.com/problems/climbing-stairs/>):

You are climbing a staircase. It takes n steps to reach the top.

Each time, you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top? Compute this in $O(n)$ time.

Example: if $n = 3$, then there are three ways to reach the top:

1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

Problem 2 (Source: <https://leetcode.com/problems/house-robber/>):

You are given an array of integers a_1, a_2, \dots, a_n .

You can pick any set of indices $S \subseteq \{1, 2, \dots, n\}$, with the constraint that no two indices in S differ by 1. Find the maximum possible value of $\sum_{i \in S} a_i$, in $O(n)$ time.

Problem 3 (Source: <https://youtu.be/YBSt1jYwVfU&t=700>):

You are climbing a staircase. It takes n steps to reach the top.

In one move, you can take either 1 or 2 steps. How many ways can you reach the top, if you can take at most k moves? Compute this in $O(nk)$ time. You may assume n and k are positive integers.

Problem 1:

(Source: <https://leetcode.com/problems/climbing-stairs/>)

You are climbing a staircase. It takes n steps to reach the top.

Each time, you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top? Compute this in $O(n)$ time.

Example: if $n = 3$, then there are three ways to reach the top:

1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

Solution to 1:

Call the choice of taking 1 or 2 steps a **move**.

Consider any sequence of moves that allows you to reach the top (n steps total). What observations can we make?

Firstly, at least one move is necessary, since otherwise we can't take any steps. So, let's look at the last move:

- If we take 1 step in the last move, then our previous moves must consist of a total of $n - 1$ steps.
- If we take 2 steps in the last move, then our previous moves must consist of a total of $n - 2$ steps.

These are the only two options for the last move, so we can classify all sequences of moves in this way.

As an example, consider $n = 4$. Broadly speaking, there are two ways to take a total of n steps:

- Choose any sequence of moves with a total of 3 steps. Then, add a final move where you take 1 step.
- Choose any sequence of moves with a total of 2 steps. Then, add a final move where you take 2 steps.

This approach works for any positive integer number of moves.

❗ Formula for counting sequences of moves.

For any integer i , let $f(i)$ be the number of sequences of moves that consist of a total of i steps. Then,

1. $f(0) = 1$.
2. If $i < 0$, then $f(i) = 0$.
3. If $i > 0$, then $f(i) = f(i - 1) + f(i - 2)$.
4. The answer to the original problem is $f(n)$.

Justification:

1. The only way to take no steps is to make no moves, which corresponds to the empty sequence.
2. It's impossible to take a negative number of steps.
3. To take $i > 0$ steps, there must be a last move. There's $f(i - 1)$ sequences where the last move is to take 1 step, and $f(i - 2)$ sequences where the last move is to take 2 steps. Verify that this makes sense even when $i - 1$ or $i - 2$ are zero or negative.
4. Recall that we are looking for all ways to take a total of n steps.

Implementation:

To compute $f(i)$ for some positive integer i , we only need to know $f(i - 1)$ and $f(i - 2)$. So, we can compute $f(0)$, $f(1)$, $f(2)$, \dots , $f(n)$ in order—at each step, we have all the information we need.

⚠ What if we have negative indices when computing $f(i) = f(i - 1) + f(i - 2)$?

Even though we mathematically defined $f(i)$ for negative values of i , it's simpler if we stick to non-negative values for array indexing.

$i = 1$ in particular is troublesome, since our recurrence gives $f(1) = f(0) + f(-1)$. So, we manually write $f(0)$ and $f(1)$, and only use the recurrence for $i \geq 2$.

Code:

```
class Solution:
    def climbStairs(self, n: int) -> int:
        f = [0] * (n+1)    # Create list to store f(0), f(1), ...,
f(n)
        f[0] = 1
        f[1] = 1
        # At this point, only f(0) and f(1) are defined,
        # and the rest of the array has garbage values.
        # That's okay, since we're computing from left to right.
        for i in range(2, n+1):
            f[i] = f[i-1] + f[i-2]
        return f[n]
```

The algorithm runs in $O(n)$ time, as required.

Problem 2:

(Source: <https://leetcode.com/problems/house-robber/>)

You are given an array of integers a_1, a_2, \dots, a_n .

You can pick any set of indices $S \subseteq \{1, 2, \dots, n\}$, with the constraint that no two indices in S differ by 1. Find the maximum possible value of $\sum_{i \in S} a_i$, in $O(n)$ time.

Solution to 2:

Let's try to construct a set S that maximizes the sum of values a_i .

- If we choose to put n in S , then the constraint requires $n - 1 \notin S$. So, the best we can do is to add all elements from a valid subset of $\{1, 2, \dots, n - 2\}$ with maximum value.
- If we choose not to put n in S , then we should take a valid subset of $\{1, 2, \dots, n - 1\}$ with maximum value.

The fundamental issue is that we can't add both n and $n - 1$; we can pick at most one. But how do we know which one to pick?

Bogus solution.

We'll store the indices chosen so far in a set S , initially empty.

- If at least two indices are remaining, call the last two $n - 1$ and n . We know we can't add both $n - 1$ and n , so we'll do this: if $a_{n-1} > a_n$, then add $n - 1$ to S ; otherwise, add n to S .
- If one index is remaining, add it to S .
- If no indices are remaining, we are done.

The optimal selection of indices is the final set, S .

This is wrong. For example, let $[a_1, a_2, a_3] = [2, 3, 2]$. This approach achieves a sum of $a_2 = 3$, but $a_1 + a_3 = 4$ is better. Here, the issue is that picking $n - 1$ means we can't pick $n - 2$.

We could try to fix this approach by comparing a_{n-1} with $a_{n-2} + a_n$, but this ignores a_3 . Maybe we actually want $a_n + a_{n-3}$, or $a_{n-1} + a_{n-3}$, or $a_{n-1} + a_{n-4} \dots$

More generally, the issue with any greedy approach is that we can't make strong claims about the maximum sum by looking at a small portion of the given array. Let's return to our original approach, where we always aim for the best possible sum of the remaining elements.

Let $f(i)$ be the maximum possible value of $\sum_{i \in S} a_i$, where $S \subseteq \{1, 2, \dots, i\}$ is a set satisfying the constraint that no two elements in S differ by 1.

How can we calculate $f(n)$?

- If we put n in S , the best value we can get is $a_n + f(n - 2)$, which is achieved taking the best possible subset of $\{1, 2, \dots, n - 2\}$ and adding n to that set.
- If we don't put n in S , the best value we can get is $f(n - 1)$, which is achieved by taking the best possible subset of $\{1, 2, \dots, n - 1\}$.

Then, the best possible sum, which is $f(n)$, must be the maximum of these two cases.

So, we have the following recurrence:

Formula for $f(n)$.

Let i be any integer. Then,

1. If $i \leq 0$, then $f(i) = 0$.
2. If $i > 0$, then $f(i) = \max\{a_i + f(i - 2), f(i - 1)\}$.
3. The answer to the original problem is $f(n)$.

Once again, we can compute $f(0), f(1), f(2), \dots, f(n)$, in that order, in $O(n)$ time. We'll also set $f(1) = a_1$ so that we don't need to worry about defining $f(-1)$ in the recurrence.

Code:

```
class Solution:
    def rob(self, nums: List[int]) -> int:
        n = len(nums)
        f = [0] * (n+1)

        f[0] = 0
        f[1] = nums[0] # Unfortunately, nums is 0-indexed!

        # Once again, the stored values for f(2), ..., f(n)
        # are garbage until they are computed.
        for i in range(2, n+1):
            f[i] = max(nums[i-1] + f[i-2], f[i-1])

        return f[n]
```

Problem 3:

(Source: <https://youtu.be/YBSt1jYwVfU&t=700>)

You are climbing a staircase. It takes n steps to reach the top.

In one move, you can take either 1 or 2 steps. How many ways can you reach the top, if you can make at most k moves? Compute this in $O(nk)$ time.

Solution to 3:

Recall that in Problem 1, we defined $f(i)$ to be the number of sequences of moves that consist of i steps in total. This made sense, since all sequences with a total of i steps were essentially the same: we added extra moves to the end of each sequence without worry.

This time, we must distinguish these sequences based on the number of moves used—taking 5 steps in 4 moves is not the same as taking 5 steps in 3 moves, since we have a different number of moves remaining. This becomes a problem when we try to add moves to the end of these sequences.

The good thing is that knowing the number of steps and moves is enough to tell us how we can extend a sequence. Can you see why?

① Formula for counting the number of sequences of moves.

For any integers i and j , let $f(i, j)$ be the number of sequences of moves containing a total of i steps and j moves. Then,

1. $f(0, 0) = 1$.
2. If $i, j > 0$, then $f(i, j) = f(i - 1, j - 1) + f(i - 2, j - 1)$.
3. If $i < 0$ or $j < 0$, then $f(i, j) = 0$.
4. If exactly one of i and j are 0, then $f(i, j) = 0$.
5. The answer to the original problem is $\sum_{j=0}^k f(n, j)$.

Justification:

1. The empty sequence works.
2. Consider any sequence of moves with i steps and j moves. If the last move is to take 1 step, then the previous moves must have $i - 1$ steps (one fewer) and $j - 1$ moves (one fewer). There are $f(i - 1, j - 1)$ such sequences.
If the last move is to take 2 steps, then the previous moves must have $i - 2$ steps (two fewer) and $j - 1$ moves (one fewer). There are $f(i - 2, j - 1)$ such sequences.

3. A negative number of steps or moves doesn't make sense.
4. The only way to have no steps is to make no moves, and vice versa. So, it doesn't make sense to have 0 steps but a nonzero amount of moves, or 0 moves but a nonzero amount of steps.
5. Recall that the problem statement says we should take at most k moves. (We can also skip $j = 0$ in the summation, since $f(n, 0) = 0$.)

Implementation:

We'll store the computed values of f in a 2D array. We need to keep track of values $f(i, j)$ where i is between 0 and n and j is between 0 and k .

The order of computation also matters: we also need to be careful to not compute $f(i, j) = f(i - 1, j - 1) + f(i - 2, j - 1)$ if either $f(i - 1, j - 1)$ or $f(i - 2, j - 1)$ have garbage values.

Many orders work, but this is what we will do:

- Compute $f(0, 0), f(0, 1), \dots, f(0, k)$.
- Compute $f(1, 0), f(1, 1), \dots, f(1, k)$.
- Compute $f(2, 0), f(2, 1), \dots, f(2, k)$.
- Repeat for each i up to n .

Since each row $f(i, 0), f(i, 1), \dots, f(i, k)$ only relies on the previous two rows, this order of computation will work. We'll just have to handle $i = 0$ and $i = 1$ manually.

```
class Solution:
    def climbStairs(self, n: int, k: int) -> int:
        f = [[0] * (k+1) for _ in range(n+1)]

        f[0][0] = 1      # f(0, j) = 0 for all other j
        f[1][1] = 1      # f(1, j) = 0 for all other j.

        for i in range(2, n+1):
            # Note that f(i, 0) = 0, so we can start at j=1
            for j in range(1, k+1):
                f[i][j] = f[i-1][j-1] + f[i-2][j-1]

        # Compute answer (you can also try `sum(f[n])`)
        ans = 0
        for j in range(k+1):
            ans += f[n][j]

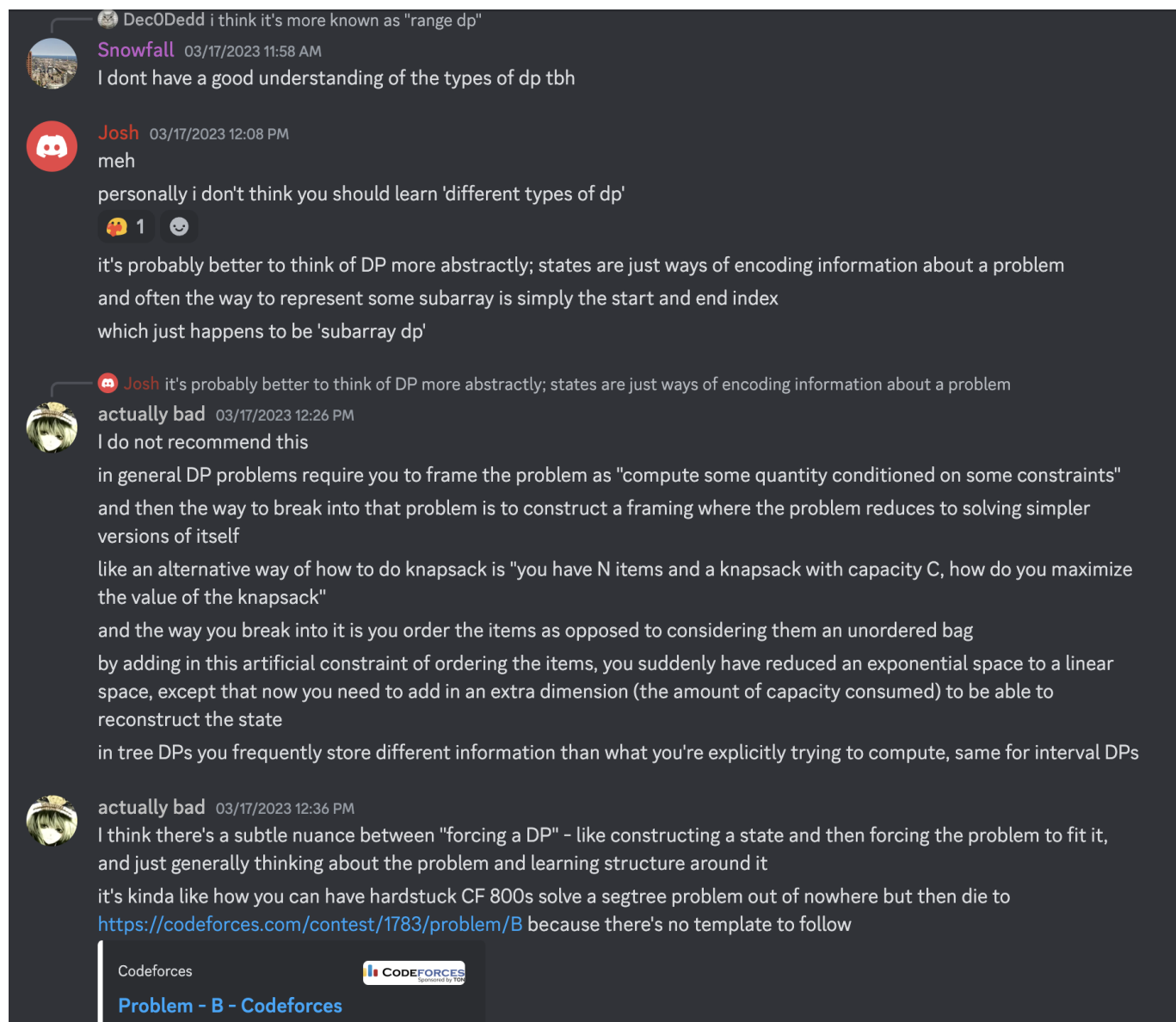
        return ans
```


It's also okay to set $f(0, 0) = 1$ and start with $i = 1$. In the recurrence, you can use if statements to check if any quantity becomes negative.

General thoughts

I chose the last problem because I think it's important to be exposed to "multidimensional" DP problems early. Adding extra dimensions sounds scary, but it really just means we are encoding more pieces of information. 1D problems are too simple to capture the full range of what DP can do.

With that being said, there's seemingly an endless supply of multidimensional DP problems to learn. Here's some advice from [Josh](#) and [xiaowuc1](#) (Discord name: "actually bad") on the DMOJ Discord server:



Dec0Dedd i think it's more known as "range dp"

Snowfall 03/17/2023 11:58 AM
I dont have a good understanding of the types of dp tbh

Josh 03/17/2023 12:08 PM
meh
personally i don't think you should learn 'different types of dp'
👍 1 🤔

it's probably better to think of DP more abstractly; states are just ways of encoding information about a problem and often the way to represent some subarray is simply the start and end index which just happens to be 'subarray dp'

Josh it's probably better to think of DP more abstractly; states are just ways of encoding information about a problem

actually bad 03/17/2023 12:26 PM
I do not recommend this
in general DP problems require you to frame the problem as "compute some quantity conditioned on some constraints" and then the way to break into that problem is to construct a framing where the problem reduces to solving simpler versions of itself
like an alternative way of how to do knapsack is "you have N items and a knapsack with capacity C, how do you maximize the value of the knapsack"
and the way you break into it is you order the items as opposed to considering them an unordered bag
by adding in this artificial constraint of ordering the items, you suddenly have reduced an exponential space to a linear space, except that now you need to add in an extra dimension (the amount of capacity consumed) to be able to reconstruct the state
in tree DPs you frequently store different information than what you're explicitly trying to compute, same for interval DPs

actually bad 03/17/2023 12:36 PM
I think there's a subtle nuance between "forcing a DP" - like constructing a state and then forcing the problem to fit it, and just generally thinking about the problem and learning structure around it
it's kinda like how you can have hardstuck CF 800s solve a segtree problem out of nowhere but then die to <https://codeforces.com/contest/1783/problem/B> because there's no template to follow

Codeforces
Problem - B - Codeforces

I'd argue that thinking about what information you need to store in states is still useful, but xiaowuc1 has a point: you need to understand the structure of the problem before you can meaningfully decide what information is necessary and sufficient for your state. There is an excellent [blog](#) by -is-this-fft- that discusses a similar idea.

The idea of **ordering** items is also powerful and applies to some non-DP problems too. For example, the problem of [interval scheduling](#) can be solved by ordering the intervals by finish time, then observing some nice properties that allow us to process intervals in order one at a time. On the other hand, if we keep the intervals out of order, it's hard to do better than trying all subsets.

Footnotes

1. I decided to skip memory optimization for all problems, since it makes the solution much less clear and requires a strong understanding of the order of computation. Usually, memory isn't an issue: if you can afford $O(n^2)$ time, you can typically afford $O(n^2)$ space too. However, it's a good idea to revisit and optimize memory for these problems once you get more experience.
2. You can solve Problem 3 in $O(n + k)$ time with some combinatorics! This is not unusual—many recurrence relations can be reduced to some form of counting.
3. In Problem 3, we assumed that the input values n and k were positive integers (otherwise, we would access the array out of bounds).