Dynamic Programming 1

Veteran Track

Gabee De Vera

What is Dynamic Programming?

What is Dynamic Programming?

Dynamic Programming is a technique where we store the answers to previous subproblems to solve the current subproblem.

For example, let's say we try to compute the nth Fibonacci number directly from the recurrence,

$$F(x) = \begin{cases} x \text{ if } 0 \leq x \leq 1 \\ F(x-1) + F(x-2) \text{ if } x > 1 \end{cases}$$

What is Dynamic Programming?

Let us try computing $F(4)$:

$$\begin{aligned} F(4) &= F(3) + F(2) \\ &= F(2) + F(1) + F(1) + F(0) \\ &= F(1) + F(0) + F(1) + F(1) + F(0) \\ &= 1 + 0 + 1 + 1 + 0 \\ &= 3 \end{aligned}$$

That wasn't too bad!

What is Dynamic Programming?

Let us try computing $F(5)$:

$$
\begin{aligned}
F(5) &= F(4) + F(3) \\
&= F(3) + F(2) + F(2) + F(1) \\
&= F(2) + F(1) + F(1) + F(0) + F(1) + F(0) + F(1) \\
&= F(1) + F(0) + F(1) + F(1) + F(0) + F(1) + F(0) + F(1) \\
&= 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 \\
&= 5
\end{aligned}
$$

Yikes! It got worse 😨

In fact, computing $F(n)$ like this takes *exponential time*. Specifically, it takes $O(\varphi^n)$ time, where $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618033$ is the Golden Ratio.

Thinking Question

How can we achieve a polynomial time solution using the given recursion?

What is Dynamic Programming?

Here's the calculation for $F(5)$ again:

$$\begin{aligned}
F(5) &= F(4) + {\color{red}F(3)} \\
&= {\color{red}F(3)} + {\color{blue}F(2)} + {\color{blue}F(2)} + F(1) \\
&= {\color{blue}F(2)} + F(1) + F(1) + F(0) + F(1) + F(0) + F(1) \\
&= F(1) + F(0) + F(1) + F(1) + F(0) + F(1) + F(0) + F(1) \\
&= 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 \\
&= 5
\end{aligned}$$

Notice that we repeat a lot of computations. If we could retrieve the answer to $F(n)$, where $n < x$ quickly, then we can potentially calculate $F(x)$ in polynomial time.

What is Dynamic Programming?

Here's the calculation for $F(5)$ again:

$$\begin{aligned}
F(5) &= F(4) + \color{red}{F(3)} \\
&= \color{red}{F(3)} + \color{blue}{F(2)} + \color{blue}{F(2)} + F(1) \\
&= \color{blue}{F(2)} + F(1) + F(1) + F(0) + F(1) + F(0) + F(1) \\
&= F(1) + F(0) + F(1) + F(1) + F(0) + F(1) + F(0) + F(1) \\
&= 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 \\
&= 5
\end{aligned}$$

Notice that we repeat a lot of computations. If we could retrieve the answer to $F(n)$, where $n < x$ quickly, then we can potentially calculate $F(x)$ in polynomial time.

Solution: **Store** the answers to previous subproblems!

What is Dynamic Programming?

Let's compute $F(5)$ this way:

$$
\begin{aligned}
F(0) &= 0 \\
F(1) &= 1 \\
F(2) &= \ ? \\
F(3) &= \ ? \\
F(4) &= \ ? \\
F(5) &= \ ?
\end{aligned}
$$

What is Dynamic Programming?

Let's compute $F(5)$ this way:

$$
\begin{aligned}
F(0) &= 0 \\
F(1) &= 1 \\
F(2) &= F(1) + F(0) \\
F(3) &= \ ? \\
F(4) &= \ ? \\
F(5) &= \ ?
\end{aligned}
$$

What is Dynamic Programming?

Let's compute $F(5)$ this way:

$$
\begin{aligned}
F(0) &= 0 \\
F(1) &= 1 \\
F(2) &= 1 \\
F(3) &= \ ? \\
F(4) &= \ ? \\
F(5) &= \ ?
\end{aligned}
$$

What is Dynamic Programming?

Let's compute $F(5)$ this way:

$$
\begin{aligned}
F(0) &= 0 \\
F(1) &= 1 \\
F(2) &= 1 \\
F(3) &= F(2) + F(1) \\
F(4) &= \ ? \\
F(5) &= \ ?
\end{aligned}
$$

What is Dynamic Programming?

Let's compute $F(5)$ this way:

$$
\begin{aligned}
F(0) &= 0 \\
F(1) &= 1 \\
F(2) &= 1 \\
F(3) &= 2 \\
F(4) &= \ ? \\
F(5) &= \ ?
\end{aligned}
$$

What is Dynamic Programming?

Let's compute $F(5)$ this way:

$$
\begin{aligned}
F(0) &= 0 \\
F(1) &= 1 \\
F(2) &= 1 \\
F(3) &= 2 \\
F(4) &= F(2) + F(3) \\
F(5) &= \ ?
\end{aligned}
$$

What is Dynamic Programming?

Let's compute $F(5)$ this way:

$$\begin{aligned}F(0) &= 0\\ F(1) &= 1\\ F(2) &= 1\\ F(3) &= 2\\ F(4) &= 3\\ F(5) &= \ ?\end{aligned}$$

What is Dynamic Programming?

Let's compute $F(5)$ this way:

$$
\begin{aligned}
F(0) &= 0 \\
F(1) &= 1 \\
F(2) &= 1 \\
F(3) &= 2 \\
F(4) &= 3 \\
F(5) &= F(3) + F(4)
\end{aligned}
$$

What is Dynamic Programming?

Let's compute $F(5)$ this way:

$$F(0) = 0$$
$$F(1) = 1$$
$$F(2) = 1$$
$$F(3) = 2$$
$$F(4) = 3$$
$$F(5) = 5$$

That was a lot faster! In fact, this takes $O(n)$ time to compute $F(n)$ at a cost of $O(n)$ *auxiliary memory*. This technique is known as **memoization** (without the r). When we **memoize** a function, we store the results of applying the function on a given set of inputs.

Implementation of Fibonacci with Memoization

```
typedef long long ll;
typedef vector<ll> vll;

const ll MAX_SIZE = 201; // Update this as needed
vll memo(MAX_SIZE, -1ll);
ll fib(ll n) {
    if(n <= 1) return n;
    ll& ans = memo[n];
    if(ans == -1) {
        ans = fib(n - 1) + fib(n - 2);
    }
    return ans;
}
```

What is Dynamic Programming?

Dynamic Programming = Brute Force + Memoization

More Common DP Problems

Minimum Coin Change

You are given a set of denominations C with values $C = \{1, c_0, c_1, c_2, c_3, \ldots, c_n\}$, in Pesos. What is the **minimum number of coins** needed to pay for an item that costs n Pesos? You may use a denomination multiple times. All denominations are integers.

Say, for example, we want to make change for an item with price $n = 30$ with denominations $C = \{1, 4, 10, 21\}$. Then, surely, the best strategy is to keep taking the largest possible denomination that fits, right? So $30 = 21 + 9 = 21 + 4 + 5 = 21 + 4 + 4 + 1$; therefore, at least 4 coins are needed to make change for $n = 30$.

Minimum Coin Change: The Greedy Approach

You are given a set of denominations C with values $C = \{1, c_0, c_1, c_2, c_3, \ldots, c_n\}$, in Pesos. What is the **minimum number of coins** needed to pay for an item that costs n Pesos? You may use a denomination multiple times. All denominations are integers.

Say, for example, we want to make change for an item with price $n = 30$ with denominations $C = \{1, 4, 10, 21\}$. Then, surely, the best strategy is to keep taking the largest possible denomination that fits, right? So $30 = 21 + 9 = 21 + 4 + 5 = 21 + 4 + 4 + 1$; therefore, at least 4 coins are needed to make change for $n = 30$.

WRONG. You can make change for $n = 30$ using only **three** coins, since $30 = 10 + 10 + 10$.

Minimum Coin Change: When Greedy Fails

What went wrong with the previous solution? Based on our real life experience, we assumed that the optimal move at any given point is to pick the largest denomination that fits, **but this is not always true**.

Minimum Coin Change: When Greedy Fails

What went wrong with the previous solution? Based on our real life experience, we assumed that the optimal move at any given point is to pick the largest denomination that fits, **but this is not always true**.

What we used is an example of a **greedy** algorithm. While they sometimes work, they oftentimes don't. Greedy algorithms need to be proven correct before usage (to be discussed in a later module). For now, here's my tip: **Greedy is bad if it's correctness is unproven**. Avoid greedy unless you have proven its correctness.

Minimum Coin Change: The Brute Force Approach

It seems like we need to somehow account for all possible ways to choose the denominations. This may seem daunting, after all, there are 20 different ways to make change for 30 pesos using denominations of $1, 4, 10,$ and 21 peso coins. This will only get worse as n becomes larger!

Minimum Coin Change: The Brute Force Approach

It seems like we need to somehow account for all possible ways to choose the denominations. This may seem daunting, after all, there are 20 different ways to make change for 30 pesos using denominations of $1, 4, 10,$ and 21 peso coins. This will only get worse as n becomes larger!

This may seem daunting, after all, there are 20 different ways to make change for 30 pesos using denominations of $1, 4, 10,$ and 21 peso coins. This will only get worse as n becomes larger!

Minimum Coin Change: The Brute Force Approach

It seems like we need to somehow account for all possible ways to choose the denominations. This may seem daunting, after all, there are 20 different ways to make change for 30 pesos using denominations of $1, 4, 10,$ and 21 peso coins. This will only get worse as n becomes larger!

This may seem daunting, after all, there are 20 different ways to make change for 30 pesos using denominations of $1, 4, 10,$ and 21 peso coins. This will only get worse as n becomes larger!

Solution: **Store** the answers to previous subproblems!

Minimum Coin Change: The DP Approach

Let's say that the smallest number of coins needed to pay for n pesos is $\mathrm{ans}(n)$. Now, let us take a leap of faith: suppose that we already *know* the answers to $\mathrm{ans}(0)$, $\mathrm{ans}(1)$, $\mathrm{ans}(2)$, ..., $\mathrm{ans}(n-1)$. Could we somehow reconstruct the answer to $\mathrm{ans}(n)$?

Minimum Coin Change: The DP Approach

Let's say that the smallest number of coins needed to pay for n pesos is $\mathrm{ans}(n)$. Now, let us take a leap of faith: suppose that we already *know* the answers to $\mathrm{ans}(0)$, $\mathrm{ans}(1)$, $\mathrm{ans}(2)$, ..., $\mathrm{ans}(n-1)$. Could we somehow reconstruct the answer to $\mathrm{ans}(n)$?

It turns out that we can! Consider the last denomination used, c. Assuming the last denomination used is c, the number of coins used to pay for n pesos is $\mathrm{ans}(n) = 1 + \mathrm{ans}(n-c)$. We can *brute force* through all possible denominations c, giving us the following recurrence,

$$\mathrm{ans}(n) = \min_{c \in C, n-c \geq 0} (\mathrm{ans}(n-c) + 1)$$

Minimum Coin Change: The DP Approach

Now, what is the base case?

Minimum Coin Change: The DP Approach

Now, what is the base case?

The base case is $\mathrm{ans}(0) = 0$, because it takes 0 coins to pay for something free!

Minimum Coin Change: The DP Approach

Now, what is the base case?

The base case is $\mathrm{ans}(0) = 0$, because it takes 0 coins to pay for something free!

What's the running time of the solution?

Minimum Coin Change: The DP Approach

Now, what is the base case?

The base case is $\mathrm{ans}(0) = 0$, because it takes 0 coins to pay for something free!

What's the running time of the solution?

We need to compute $\mathrm{ans}(x)$ for $0 \leq x \leq n$. Also, to compute each $\mathrm{ans}(x)$, we need to sum at most $|C|$ values, representing all possible last denominations that could have been picked. Thus, the running time of the solution is $O(n|C|)$.

Minimum Coin Change: Top-down Implementation

```cpp
typedef vector<int> vi;
const int MAX_MEMO = 200'001;
// numeric_limits<int>::max() acts as "positive infinity"
vi memo(MAX_MEMO, numeric_limits<int>::max());
int min_change(const vi& c, int n) {
    if(n == 0) return 0;
    int& ans = memo[n];
    if(ans == numeric_limits<int>::max()) {
        for(int cur_denom : c) {
            // Skip the current denomination if it doesn't fit
            if(n - cur_denom < 0) continue;
            ans = min(ans, min_change(c, n - cur_denom) + 1);
        }
    }
    return ans;
}
```

Minimum Coin Change: Bottom-up Implementation

```cpp
typedef vector<int> vi;

// ... Inside the Driver Function
vi ans(n + 1, numeric_limits<int>::max());
ans[0] = 0;
for(int i = 1; i <= n; i++) {
    for(int c : denoms) if(i - c >= 0) ans[i] = min(ans[i], ans[i - c] + 1);
}
```

Bitmask DP

Bitmask DP

Motivating problem: how many ways are there to tile an $N \times M$ grid with identical 1×2 and 2×1 dominoes, where $1 \leq M \leq 1000$, and $1 \leq N \leq 10$?

Bitmask DP

Motivating problem: how many ways are there to tile an $N \times M$ grid with identical 1×2 and 2×1 dominoes, where $1 \leq M \leq 1000$, and $1 \leq N \leq 10$?

Note that, since dominoes occupy 2 grid tiles each, there must be an even number of tiles. Otherwise, it is impossible to tile the grid.

Bitmask DP

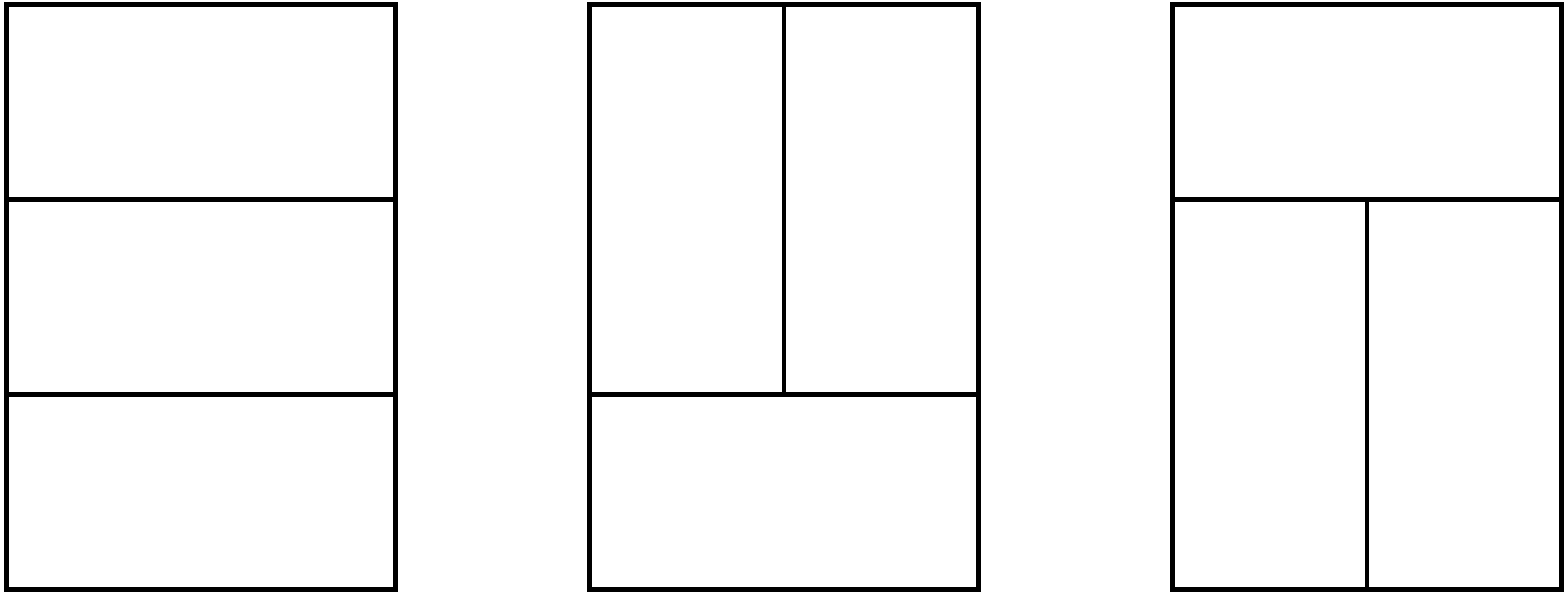
Motivating problem: how many ways are there to tile an $N \times M$ grid with identical 1×2 and 2×1 dominoes, where $1 \leq M \leq 1000$, and $1 \leq N \leq 10$?

Note that, since dominoes occupy 2 grid tiles each, there must be an even number of tiles. Otherwise, it is impossible to tile the grid.

For now, we will consider the case $N = 3$. The general case of $1 \leq N \leq 10$ will be left as an exercise.

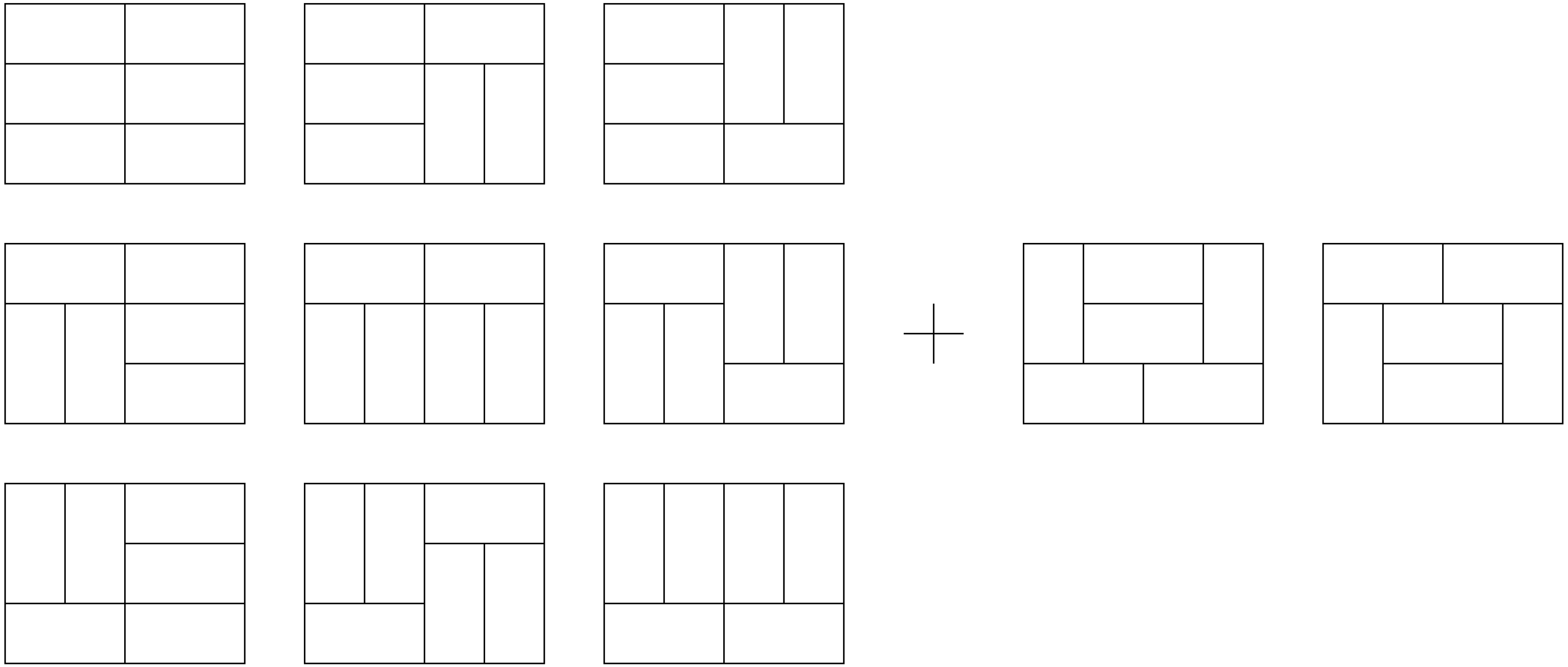
Bitmask DP

Let us first try small cases. Consider $N = 3$ and $M = 2$. There are 3 possible tilings, shown below.



Bitmask DP

Let us first try small cases. Consider $N = 3$ and $M = 4$. There are 11 possible tilings, shown below.



Bitmask DP

What about when $N = 3$ and $M = 0$? How many tilings are there?

Bitmask DP

What about when $N = 3$ and $M = 0$? How many tilings are there?

There is Exactly ONE Tiling (The Empty Tiling)

Bitmask DP

Now, let us try to find a DP solution to this problem.

What DP state should we use? This turns out to be nontrivial, so feel free to think about it for a moment.

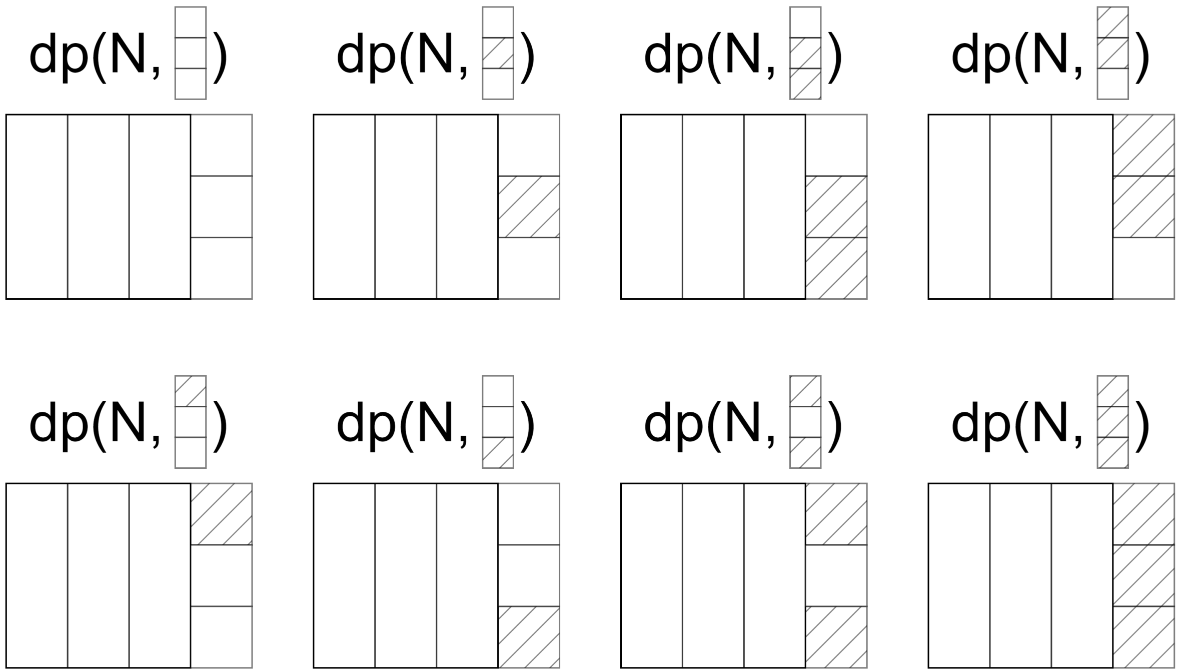
Bitmask DP

Now, let us try to find a DP solution to this problem.

What DP state should we use? This turns out to be nontrivial, so feel free to think about it for a moment.

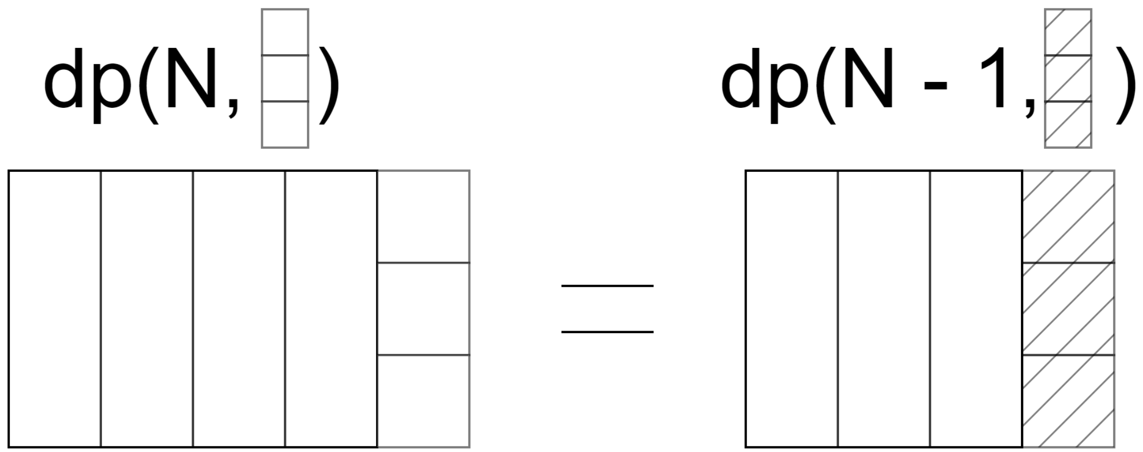
Bitmask DP

Notice that you can **perform casework on the final column**. You can assume that some tiles in the final column are currently occupied, while some are unoccupied, like so.



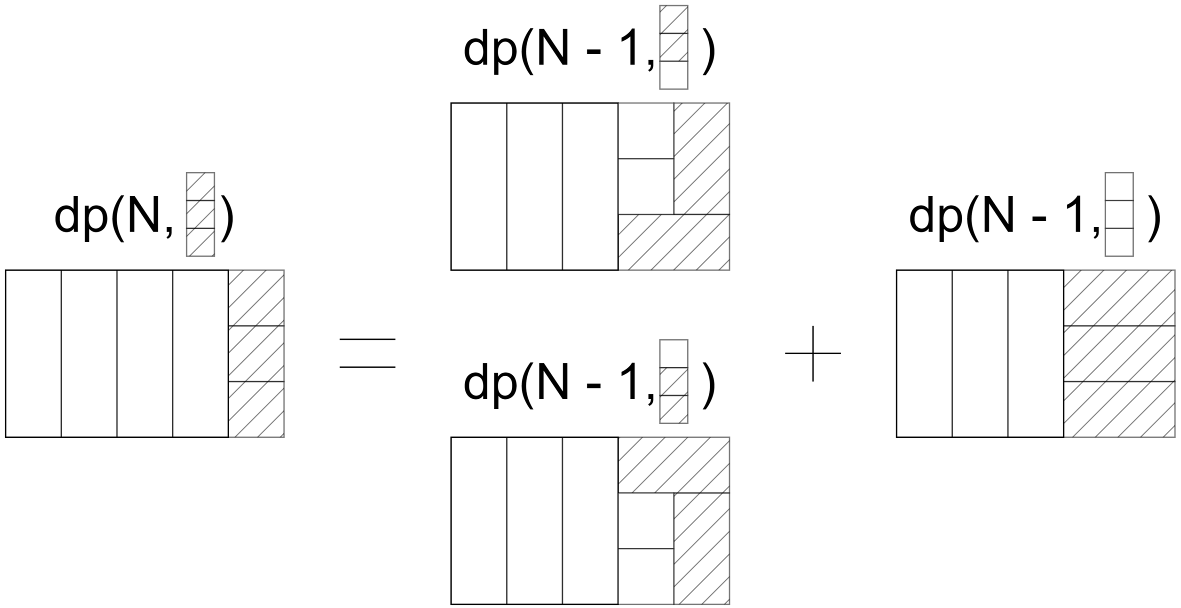
Bitmask DP

Then, for each possible state of the final row, we can do some casework to determine how it relates to smaller cases.



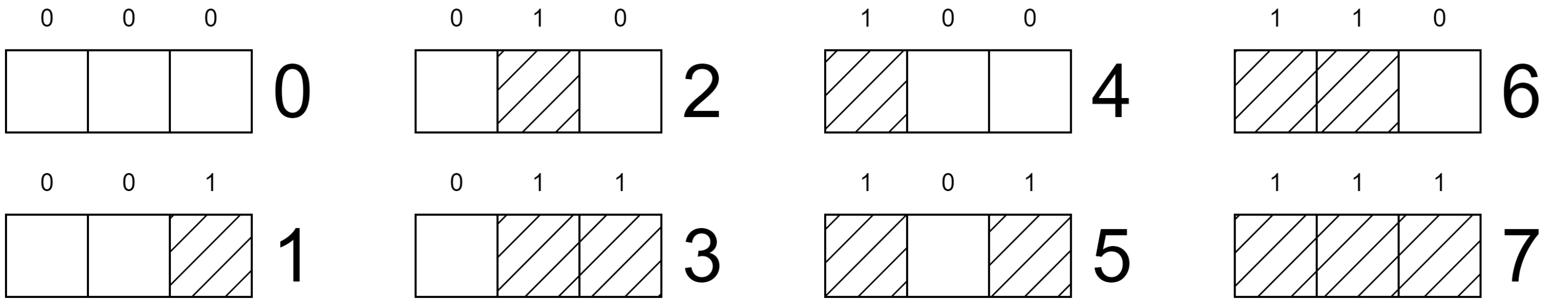
Bitmask DP

Then, for each possible state of the final row, we can do some casework to determine how it relates to smaller cases.



Bitmask DP

But now, how do we represent the state of the last column in code? Notice that each tile in the last column could either be **occupied** or **not occupied**.
Thus, there is a **binary choice** per tile of whether or not it should be occupied. We can then associate each ending state with a unique integer, like so.



This integer is known as a **bitmask**. Bitmasks are a fancy way of storing an array of booleans in a single number.

Bitmask DP

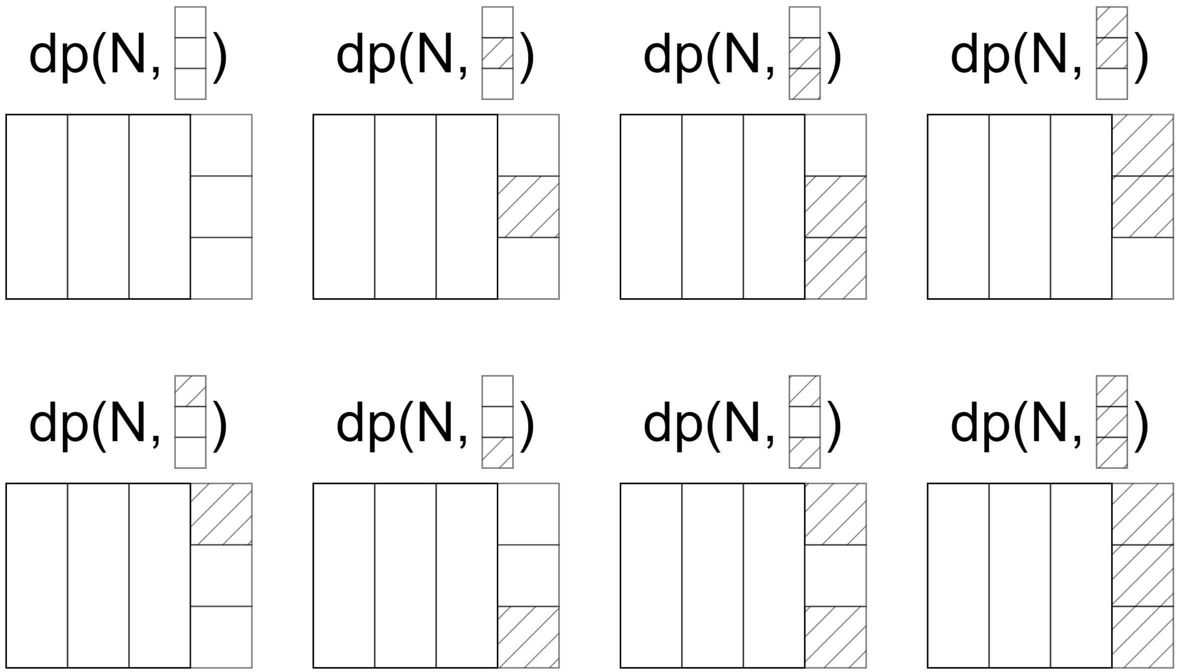
Bitmasks have a lot of advantages. For instance, you can use bitmask operations such as bitwise right shift and bitwise AND to extract the nth bit of a bitmask.

$$\begin{aligned}
\text{3rd bit of } 11010_2 &= (11010_2 >> 2)\&1_2 \\
&= (110_2)\&1_2 \\
&= 0_2
\end{aligned}$$

See this video for more information on bitmasks.

Bitmask DP

Now, all that's left to do is to complete the casework on all other possible states of the last column. You also need to determine the initial values of the DP. This will be left as an exercise for you to accomplish.



Bitmask DP

To generalize to $1 \leq N \leq 10$, you have to **programmatically** perform your DP transition (please do not do the DP transitions manually ;-;).

The time complexity of this solution is $O(M2^N)$, which should pass.

DP is Powerful

Things DP can Do

1. It can turn certain brute force solutions, which may run in exponential time, to something that runs in polynomial time.

Things DP can Do

1. It can turn certain brute force solutions, which may run in exponential time, to something that runs in polynomial time.
2. It can turn solutions that run in factorial time to solutions that run in exponential time.

Things DP can Do

1. It can turn certain brute force solutions, which may run in exponential time, to something that runs in polynomial time.
2. It can turn solutions that run in factorial time to solutions that run in exponential time.
3. It can compute the nth term of a linear recurrence in $O(\log n)$ time. In particular, it allows you to calculate the nth Fibonacci number in $O(\log n)$ time. More on this next week!

Things DP can Do

1. It can turn certain brute force solutions, which may run in exponential time, to something that runs in polynomial time.
2. It can turn solutions that run in factorial time to solutions that run in exponential time.
3. It can compute the nth term of a linear recurrence in $O(\log n)$ time. In particular, it allows you to calculate the nth Fibonacci number in $O(\log n)$ time. More on this next week!
4. Etc...

Homework

See the Reboot Page for your homework this week 😄