Fibonacci Revisited

Recursive Fibonacci Algorithm

Recall the Fibonacci sequence $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

$$F_n = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

This mathematical definition led us to a recursive algorithm

Recursive Fibonacci

$$F_n = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

```
int64_t recursiveFibonacci(int n) {
  if (n == 0 || n == 1) {
    return n;
  }
  return recursiveFibonacci(n-1) + recursiveFibonacci(n-2);
}
```

Benchmark

```
Benchmark                                    Time
-------------------------------------------------
recursiveFibonacciBench/45                 3.47 s
iterativeFibonacciBench/45                0.129 us
```


fib(5)
fib(4)
fib(3)
fib(3)
fib(2)
fib(1)
fib(2)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)
fib(1)
fib(0)

Number of Leaves

Let C_n be the number of leaves in this computation tree on input n.

This is the number of times we evaluate the base cases $\texttt{fib}(0)$ and $\texttt{fib}(1)$.

In the base cases $n = 0$ and $n = 1$ there is just one leaf so $C_n = 1$.

Otherwise, the number of leaves in the tree for $\texttt{fib}(n)$ is the sum of the number of leaves in the trees for $\texttt{fib}(n-1)$ and $\texttt{fib}(n-2)$.

$$C_n = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ C_{n-1} + C_{n-2} & \text{otherwise} \end{cases}$$

Number of Leaves

$$C_n = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ C_{n-1} + C_{n-2} & \text{otherwise} \end{cases}$$

$$F_n = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

These just differ in the base case: $C_0 = F_1, C_1 = F_2$.

So we have $C_n = F_{n+1}$.

The number of leaves grows like the Fibonacci numbers, which is very fast!

$$F_n \approx \frac{1.61^n}{\sqrt{5}}$$

Improving the Algorithm

If we have already computed something, remember the answer.

Then we don't have to compute it again.

Remembering the answer is called memoization (like writing a memo).

```
class Fibonacci {
 private:
  // memo will store the values we have computed
  std::vector<int64_t> memo {};
  int64_t fibonacciHelper(int n);
 public:
  Fibonacci();
  int64_t compute(int n);
};
```

Improving the Algorithm

```
int64_t Fibonacci::fibonacciHelper(int n) {
  // if memo.at(n) >= 0 we have already computed fib(n)
  if (memo.at(n) >= 0) {
    return memo.at(n);
  }
  memo.at(n) = fibonacciHelper(n - 1) + fibonacciHelper(n - 2);
  return memo.at(n);
}

int64_t Fibonacci::compute(int n) {
  if (n <= 1) {
    return n;
  }
  // use -1 to indicate we have not yet computed fib(n)
  memo.resize(n + 1, -1);
  memo.at(0) = 0;
  memo.at(1) = 1;
  return fibonacciHelper(n);
}
```

Improving the Algorithm

```
---------------------------------------------------------
Benchmark                                          Time
---------------------------------------------------------
iterativeFibonacciBench/70                     0.171 us
memoizedFibonacciBench/70                      0.508 us
```

With memoization the running time of the recursive algorithm to compute the 70th Fibonacci number is now under a microsecond.

Dynamic Programming

This is our first example of dynamic programming:

Like divide and conquer we express the solution to the original problem in terms of similar subproblems.

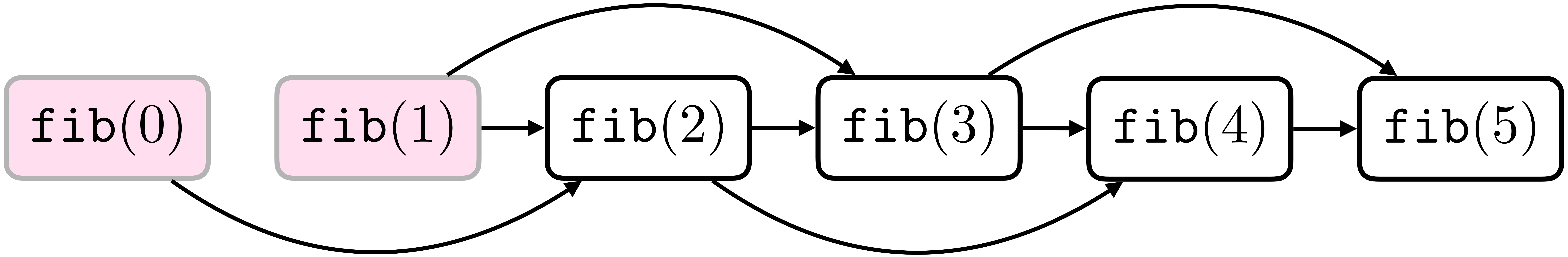
Unlike divide and conquer, a hallmark of dynamic programming is overlap between subproblems.

With this recursion + memoization approach we avoid duplicating work.

Iterative + Topo Sort

There is another way we can view dynamic programming.

Look at the dependency graph of the subproblems.

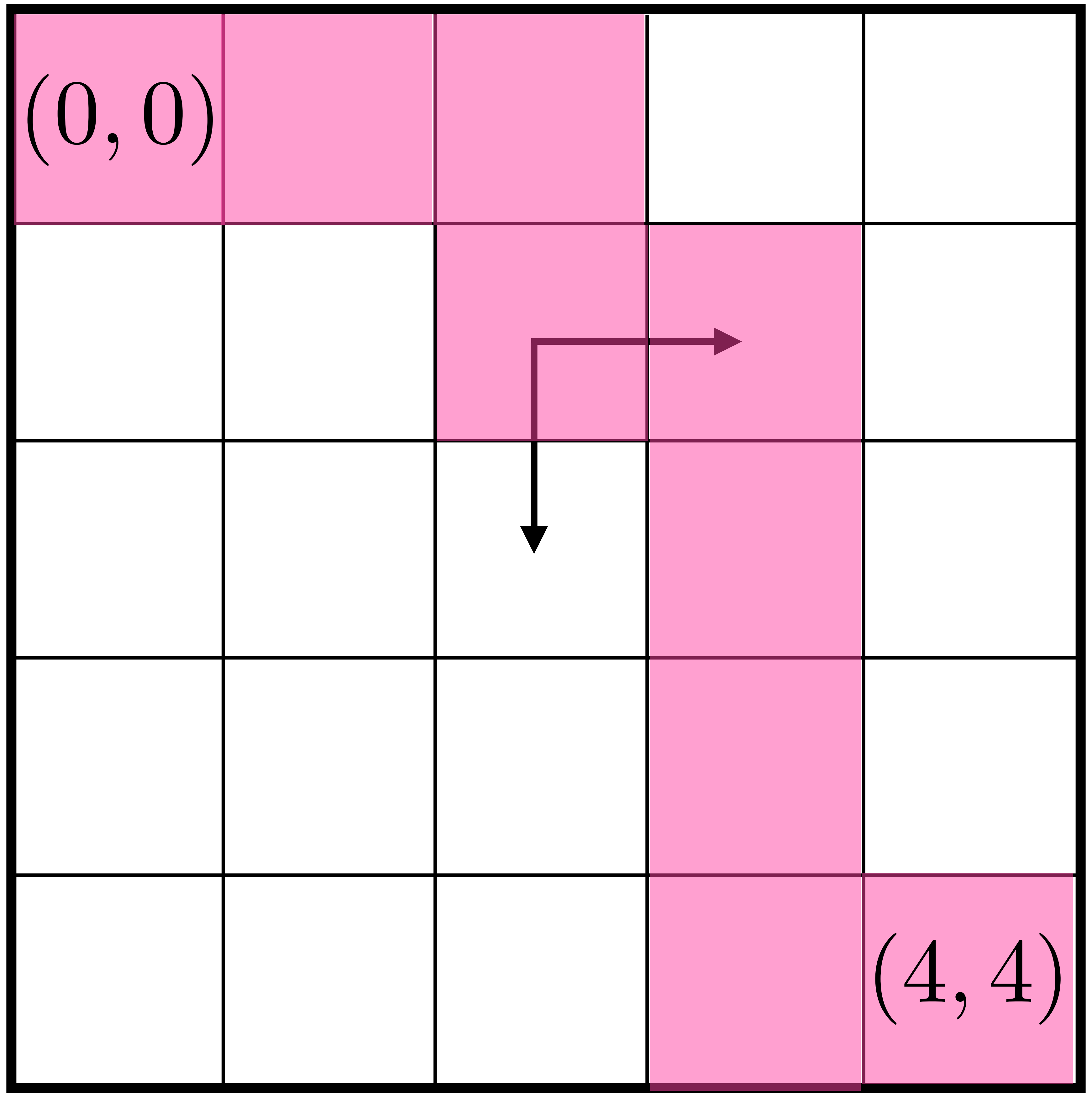


Do a topological sort of this graph, and solve the subproblems in this order.

Example: Counting Paths

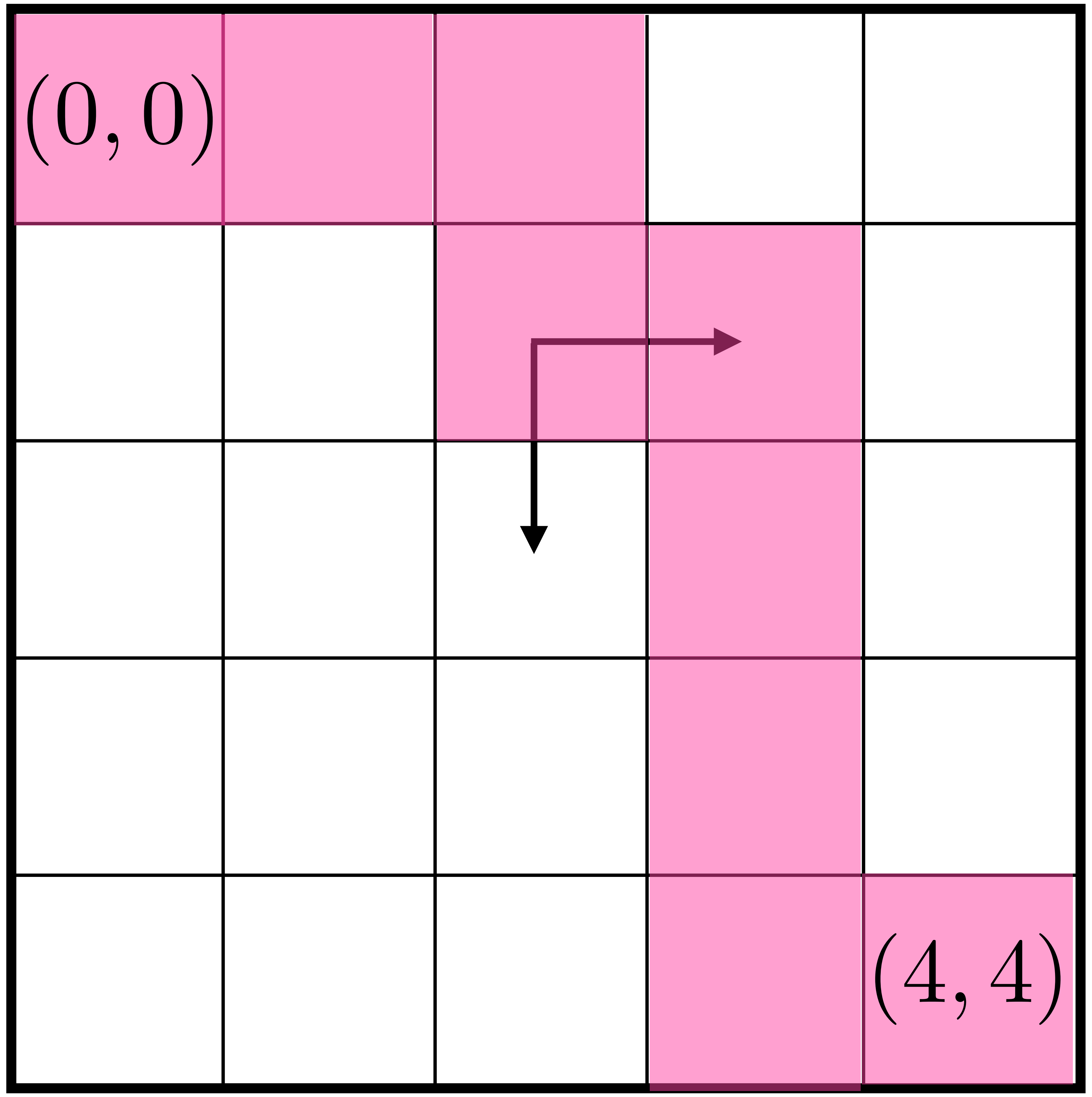
Counting Paths in Grid

How many paths from the top left corner to the bottom right corner when from each cell we can move either down or to the right?



Dynamic Programming

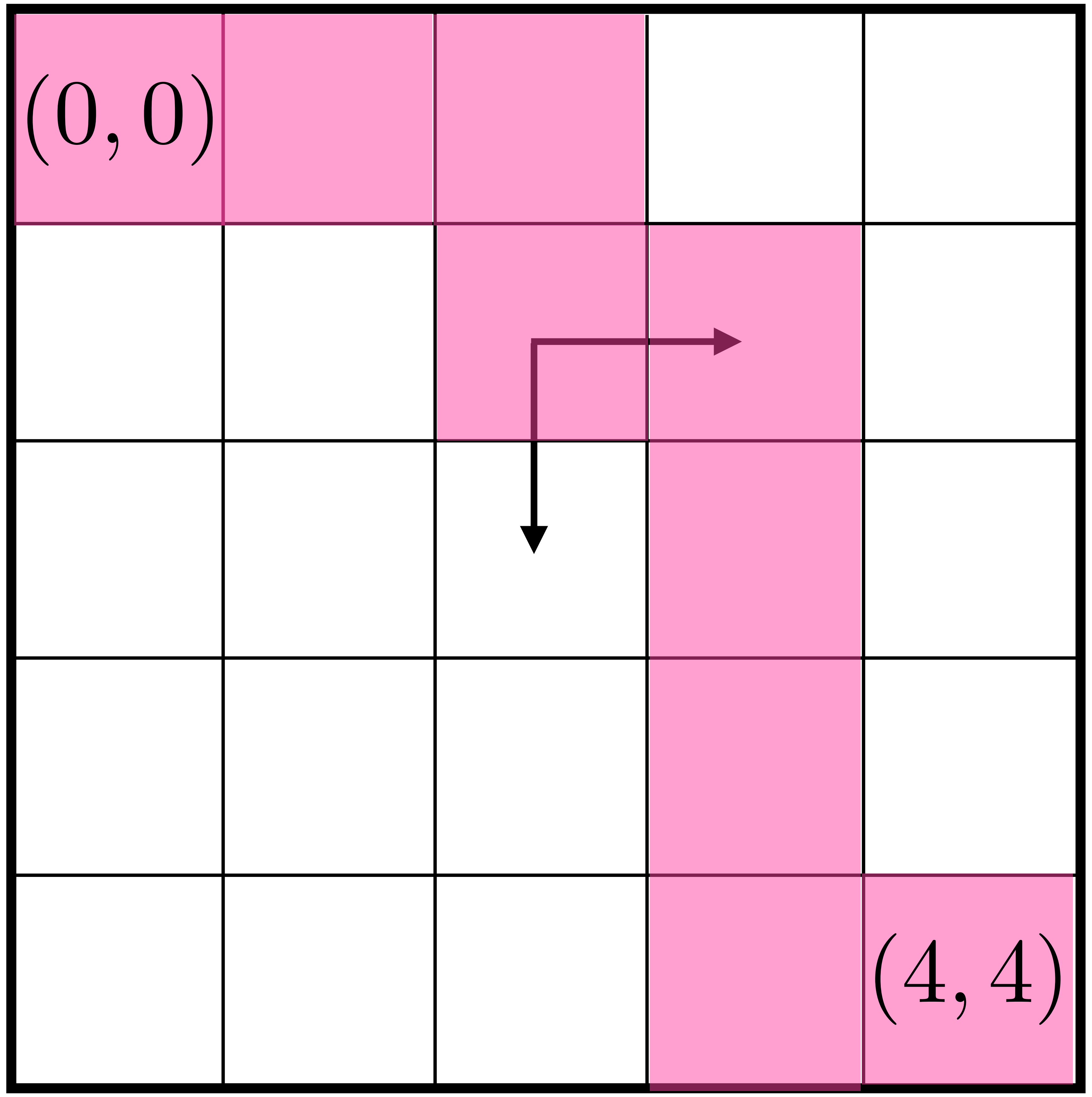
Step 1: Decide on the subproblems.



Original Problem: number of paths from $(0, 0)$ to $(4, 4)$.

Dynamic Programming

Step 1: Decide on the subproblems.

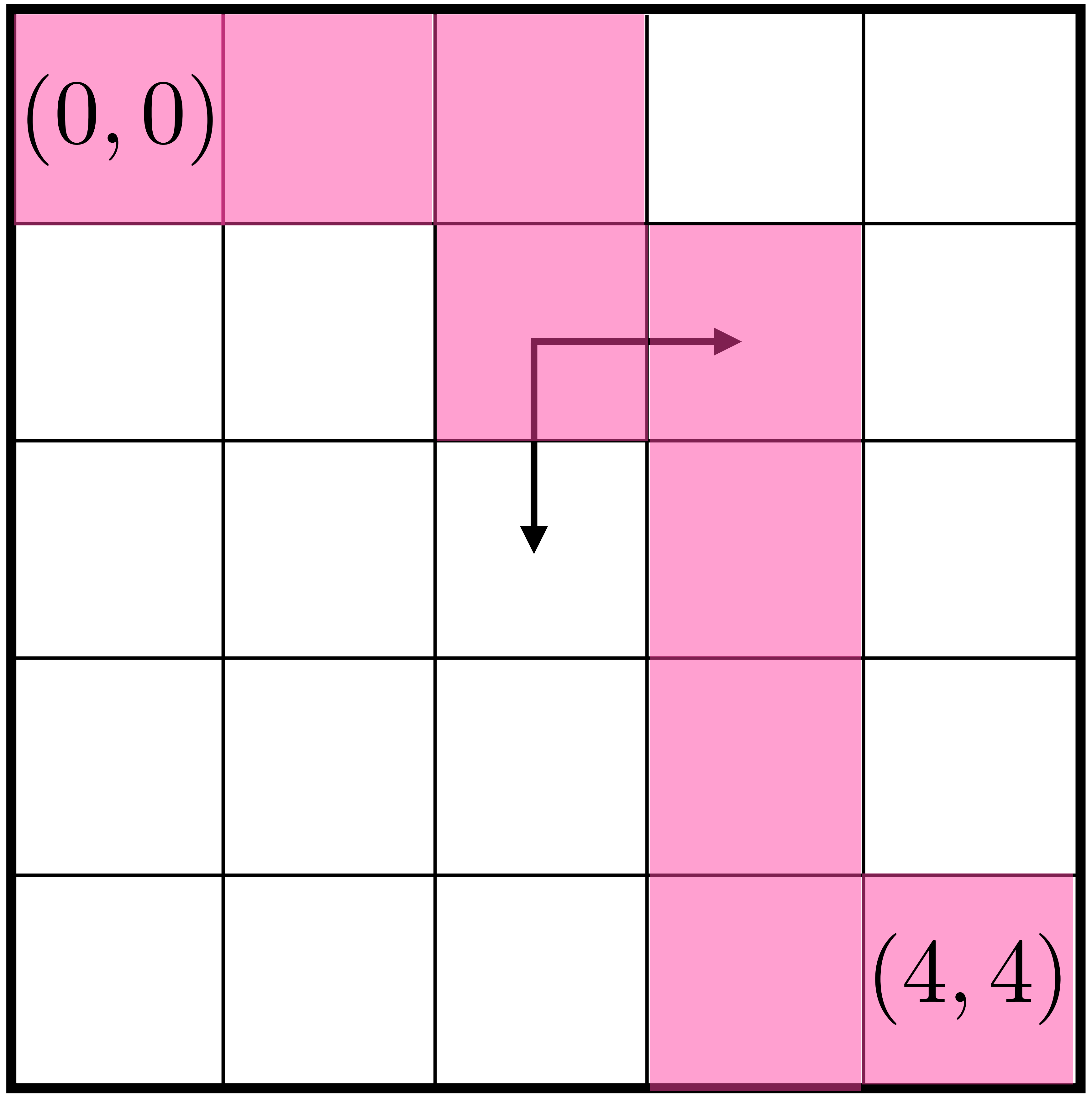


Original Problem: number of paths from (0, 0) to (4, 4).

It looks like knowing the number of paths from (0, 0) to (4, 3) would be useful!

Dynamic Programming

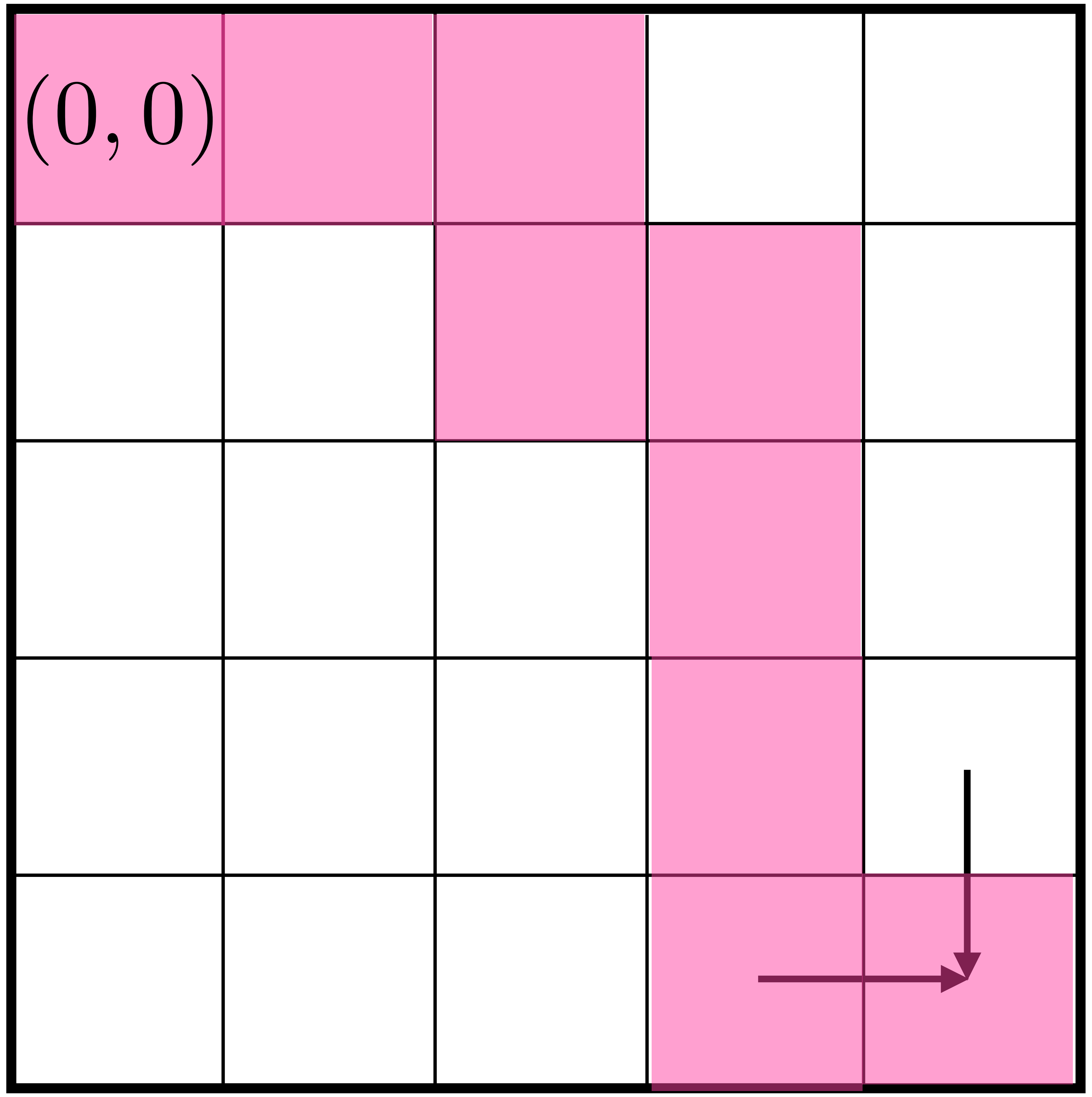
Step 1: Decide on the subproblems.



Subproblem: number of paths from (0, 0) to (x, y).

Dynamic Programming

Step 2: Develop recurrence relation.



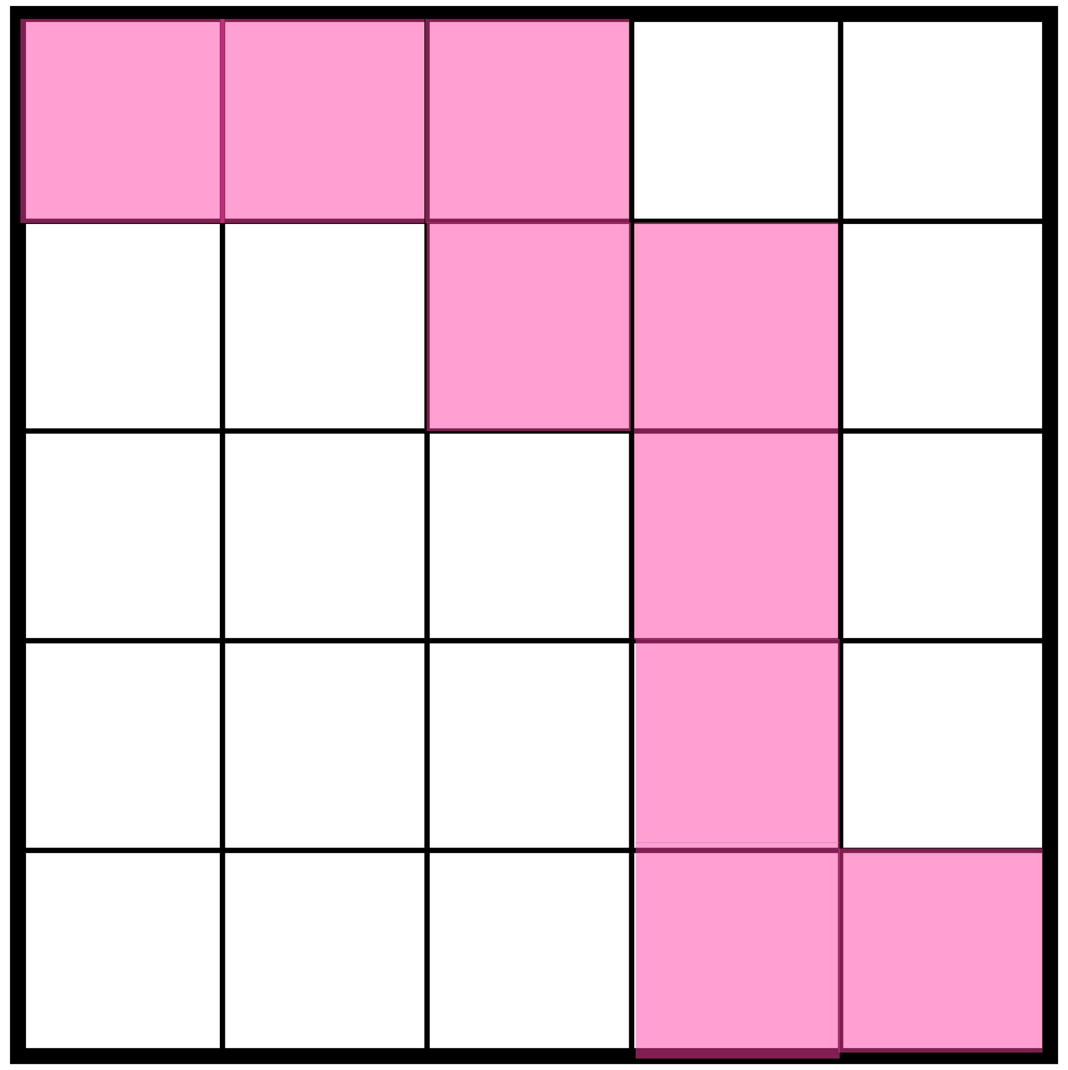
To arrive at (4, 4) we have to come from the north or west.

$$\texttt{numPathsTo}(4,4) = \texttt{numPathsTo}(3,4) + \texttt{numPathsTo}(4,3)$$

Dynamic Programming

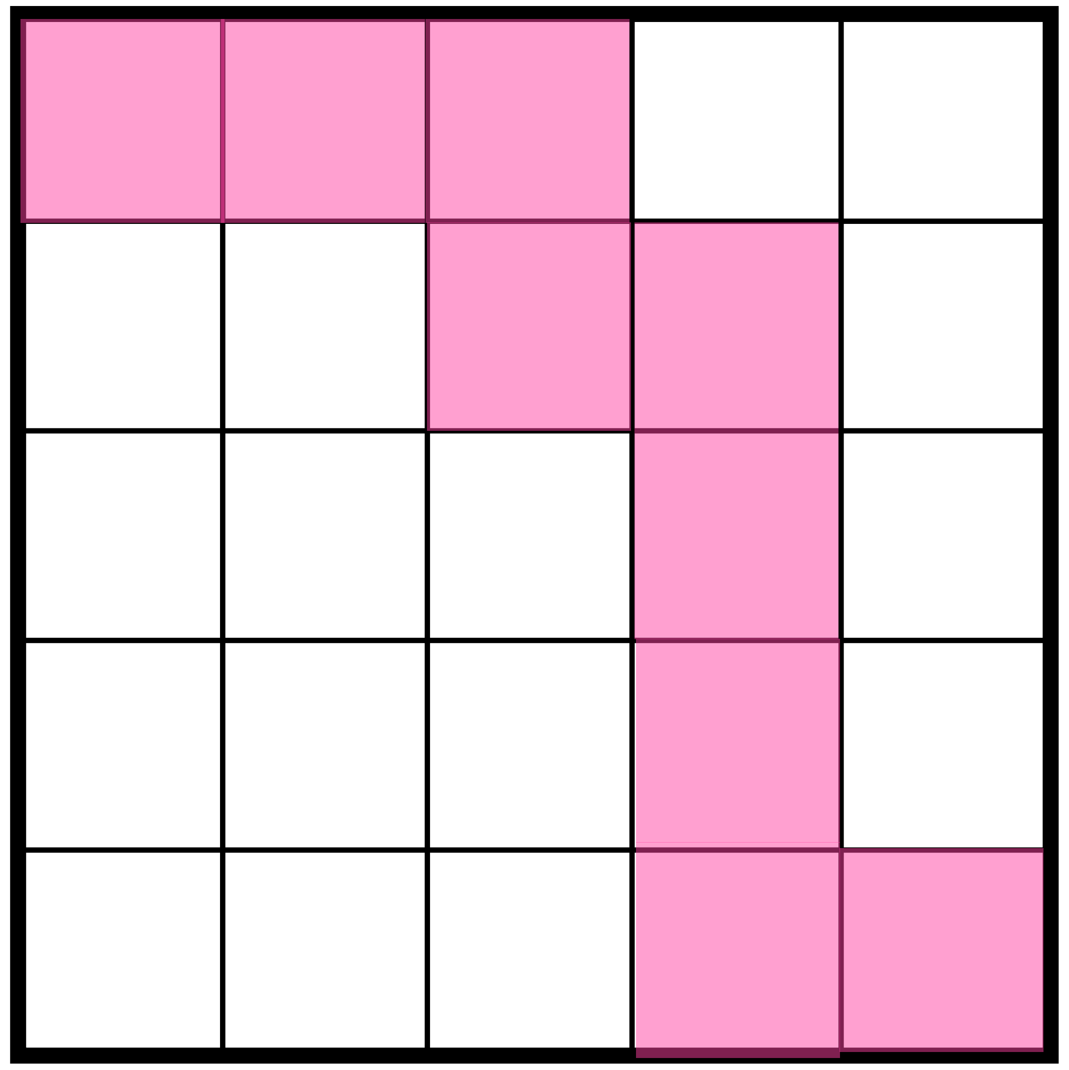
Step 2: Develop recurrence relation.

$$\texttt{numPathsTo}(x, y) = \begin{cases} 1 & \text{if } x = 0 \text{ or } y = 0 \\ \texttt{numPathsTo}(x-1, y) & \\ + \texttt{numPathsTo}(x, y-1) & \text{otherwise} \end{cases}$$



Dynamic Programming

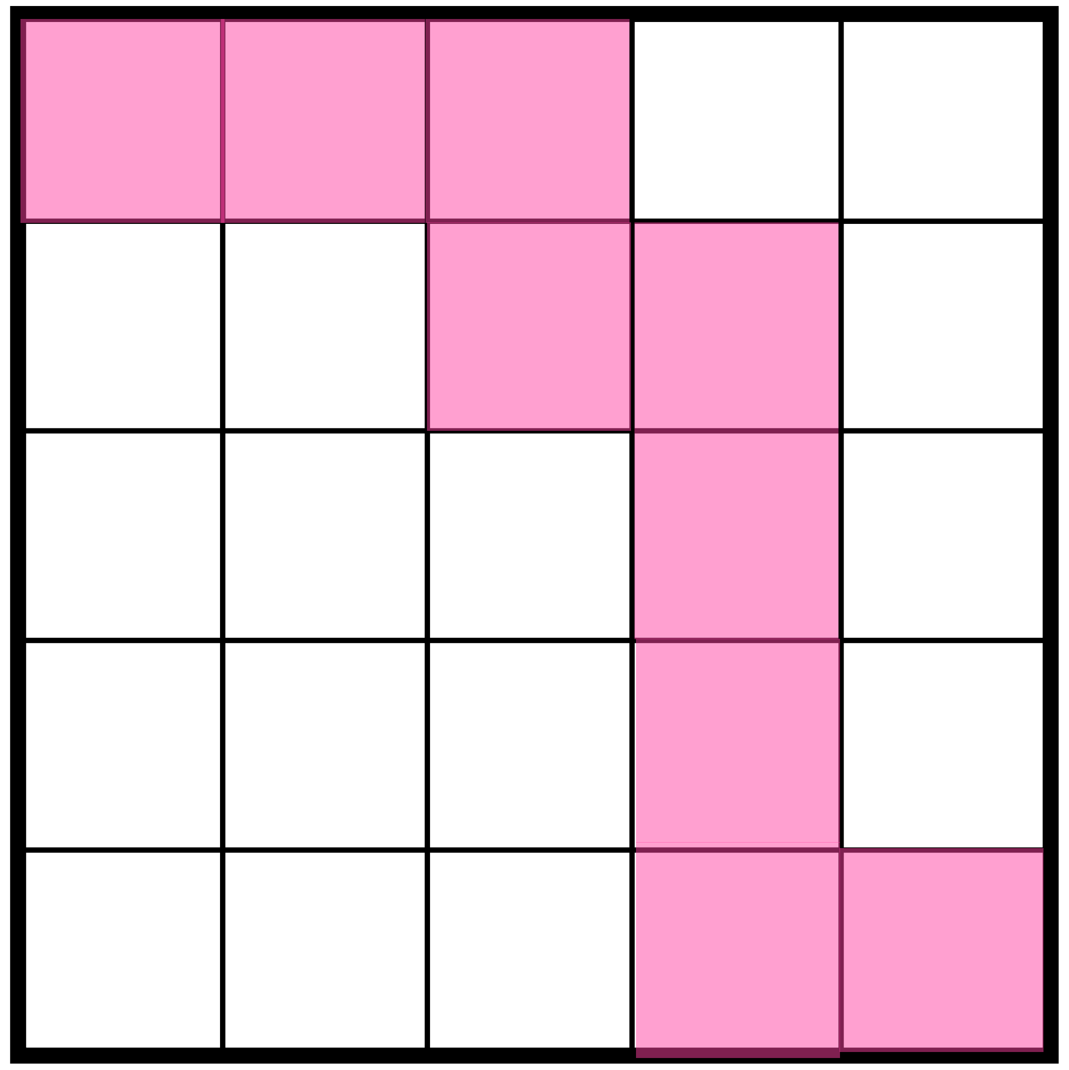
Step 3: Option 1 is to use recursion with memoization



```
std::map<std::pair<int, int>, int> numPathsTo {};
// populate memo table with base cases
numPathsTo[{0, 0}] = 1;
for (int i = 1; i < n; ++i) {
    numPathsTo[{0, i}] = 1;
    numPathsTo[{i, 0}] = 1;
}
```

Dynamic Programming

Step 3: Option 1 is to use recursion with memoization

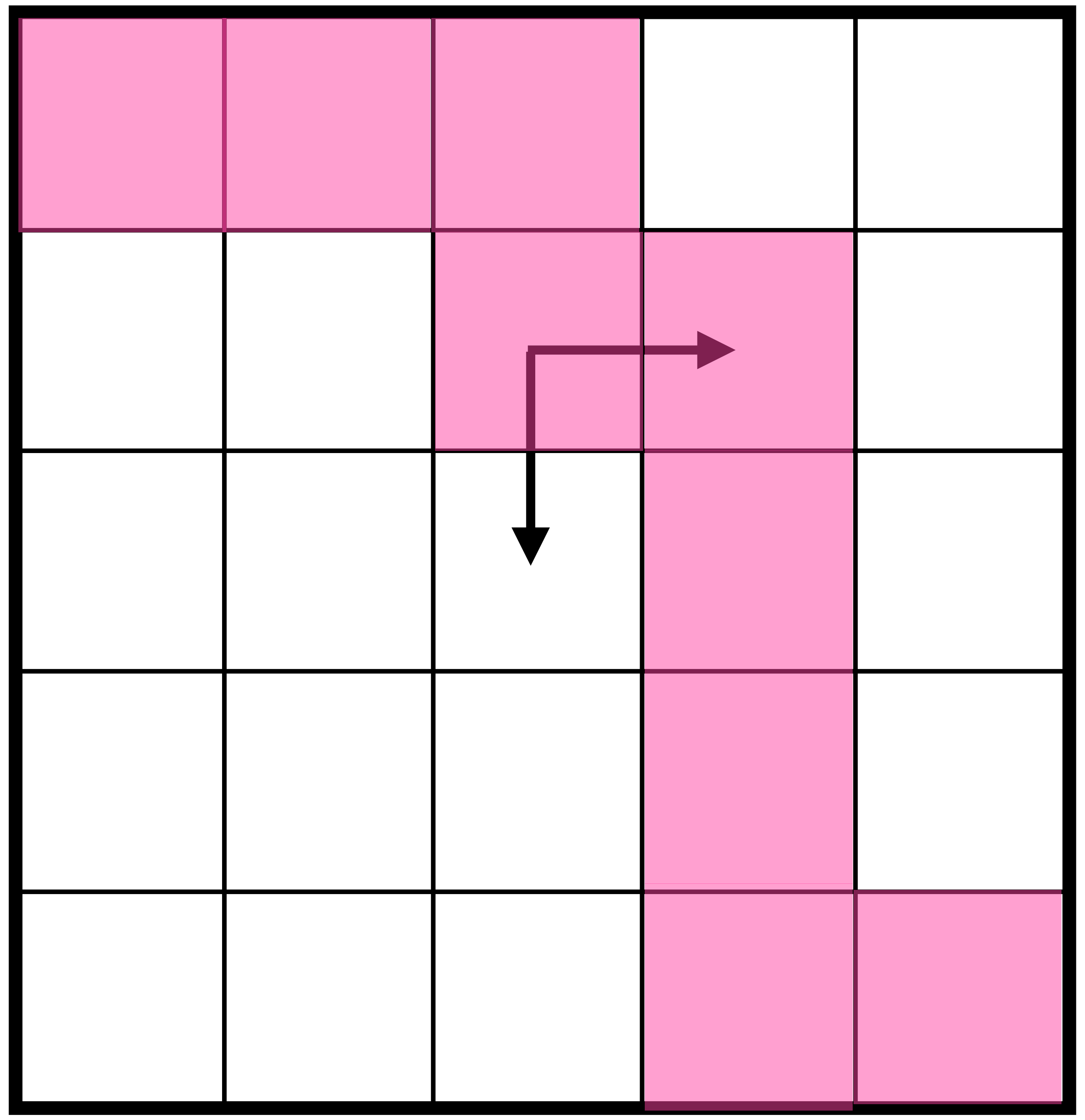


```
int countPaths(std::pair<int, int> point) {
  // if we have already computed result, use it
  if (numPathsTo.contains(point)) {
    return numPathsTo[point];
  }
  // add result to memo table and return it
  return numPathsTo[point] = countPaths({point.first - 1, point.second})
                            + countPaths({point.first, point.second - 1});
}
```

Dynamic Programming

Step 3: Option 2 is the iterative approach with topological ordering.

In what order can we solve the subproblems so that we have already have the information we need to solve the current subproblem?



Before we solve problem (x, y) we need to have already solved problems (x-1, y) and (x, y-1).

Dynamic Programming

Step 3: Option 2 is the iterative approach with topological ordering

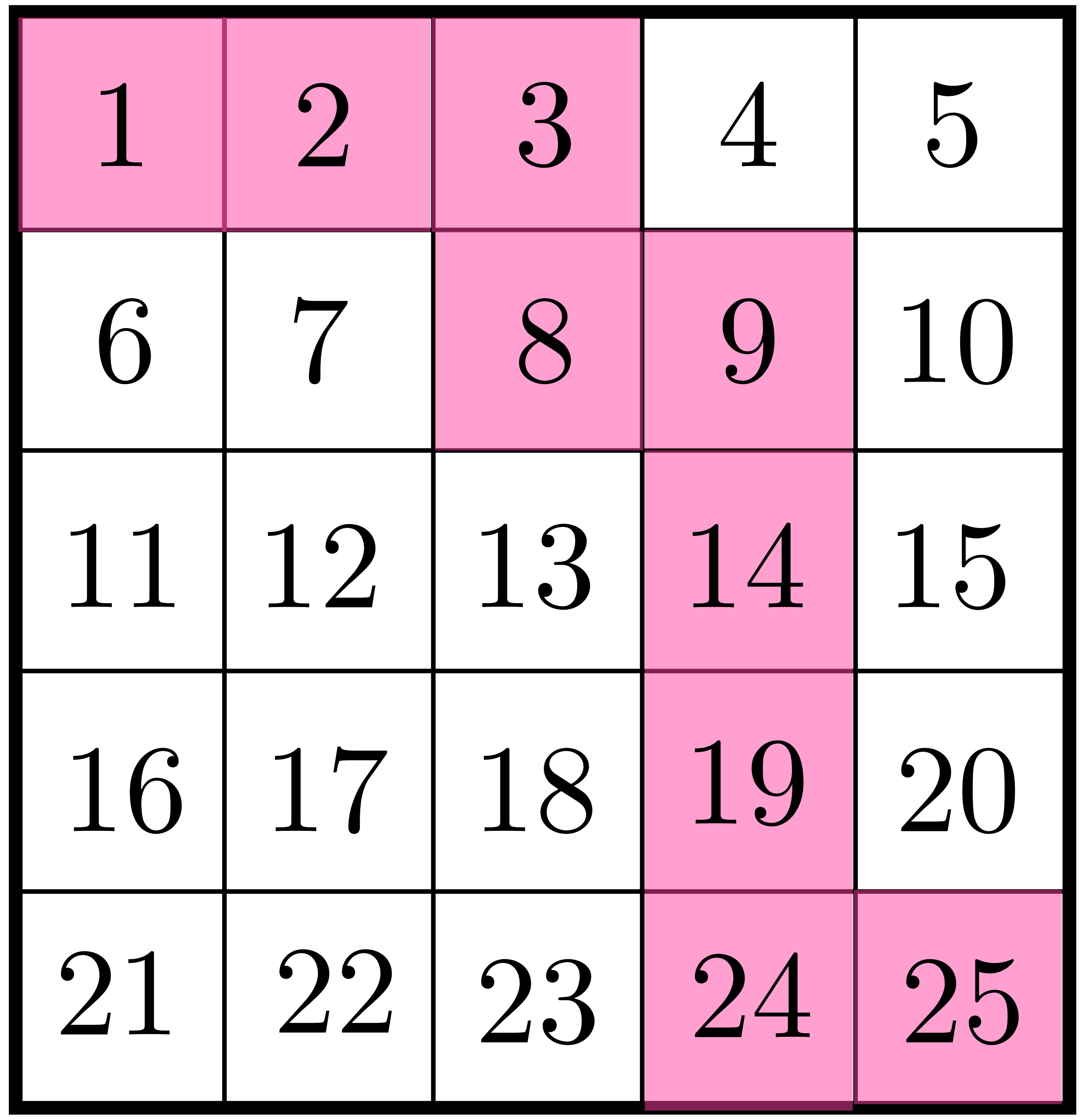
Usually the order in which to do the subproblems can be easily seen.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

In this case we can go row-by-row.

Dynamic Programming

Step 3: Option 2 is iterative approach with topological sort




```
int iterativeCountPaths(int n) {
  std::vector<int> numPathsTo(n * n);
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      if (i == 0 or j == 0) {
        numPathsTo.at(i * n + j) = 1;
      } else {
        numPathsTo.at(i * n + j) = numPathsTo.at((i - 1) * n + j)
                                 + numPathsTo.at(i * n + j - 1);
      }
    }
  }
  return numPathsTo.back();
}
```