CS4054
Bioinformatics

Spring 2025

Rushda Muneer

Introduction to Dynamic Programming

- Recursive Fibonacci Numbers
- If Fib(n) is the n-th Fibonacci number, then
  - Fib(n) = Fib(n – 1) + Fib(n – 2)
- Recurrence relation:
- An expression for a function f(x) in terms of values of f(y) where y < x.
- Exercise: Write pseudocode for a recursive function that takes an integer n as an argument and returns the n-th Fibonacci number. Assume 0-based indexing.

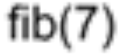
Recursive Fibonacci Numbers

- RecFib(n)

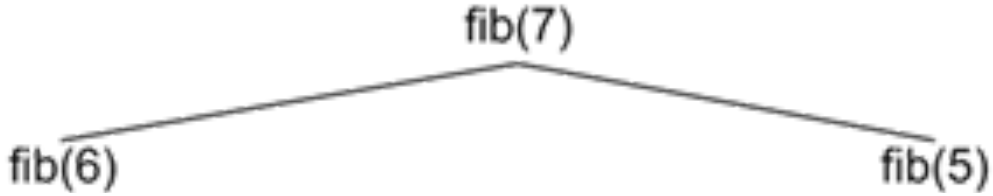
```
if n = 0 or n = 1
        return 1
else
        return RecFib(n-1) + RecFib(n-2)
```

- Is this a good algorithm? Why or why not?

Calling Fib(7) Shows the Problem with Using Recursion

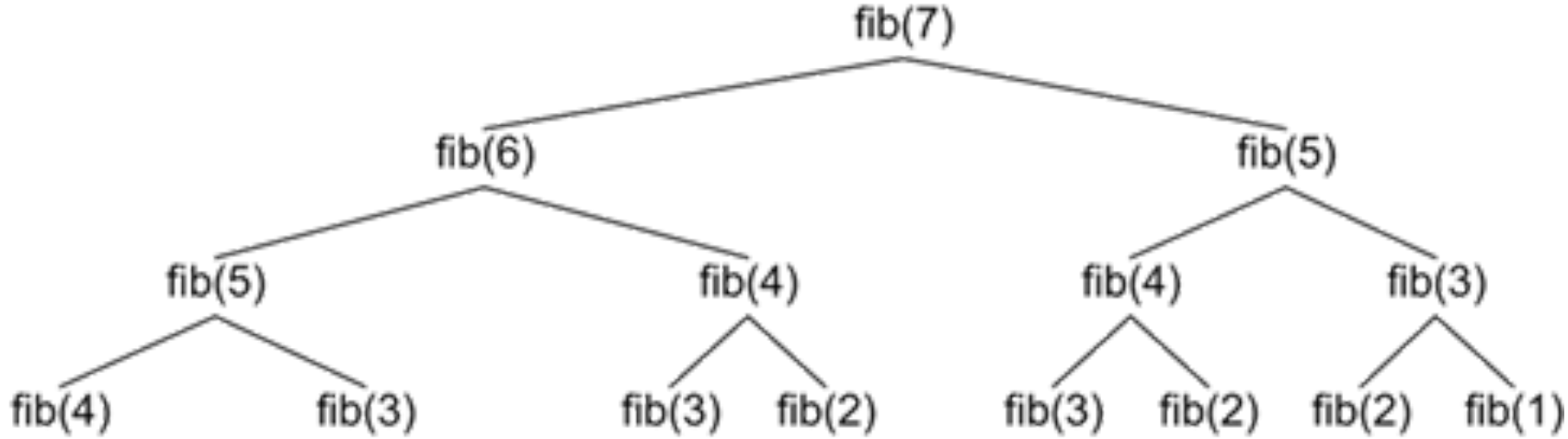


Calling Fib(7) Shows the Problem with Using Recursion

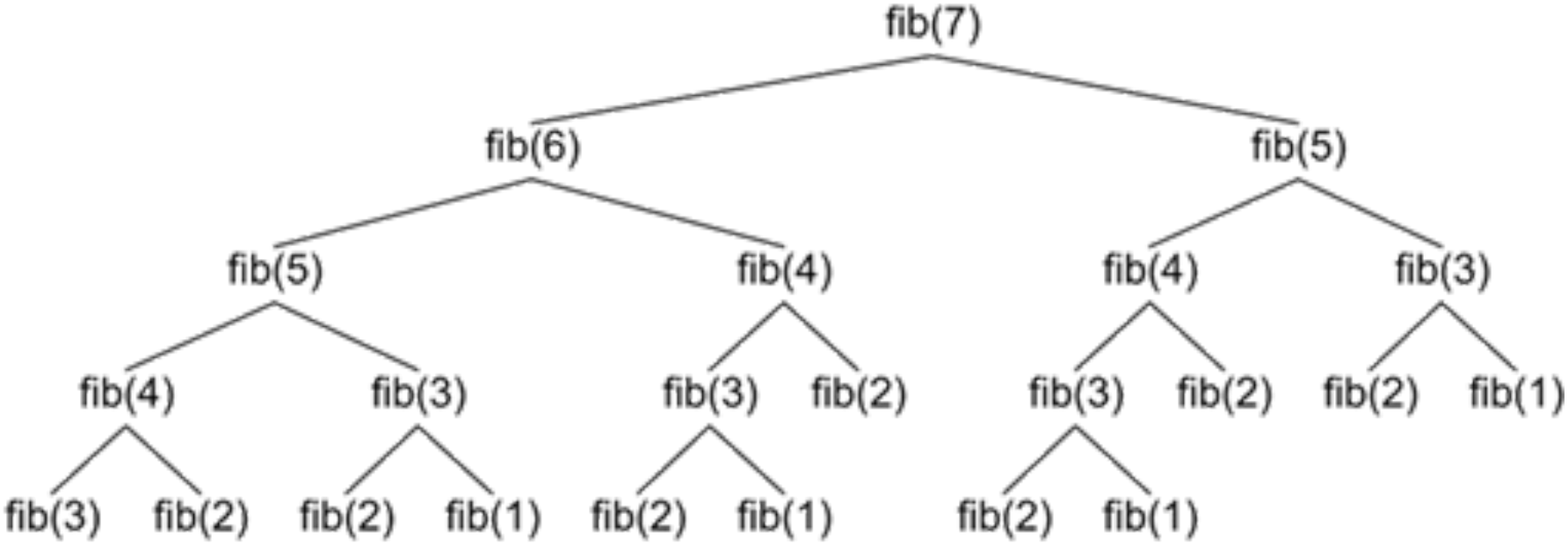


Calling Fib(7) Shows the Problem with Using Recursion

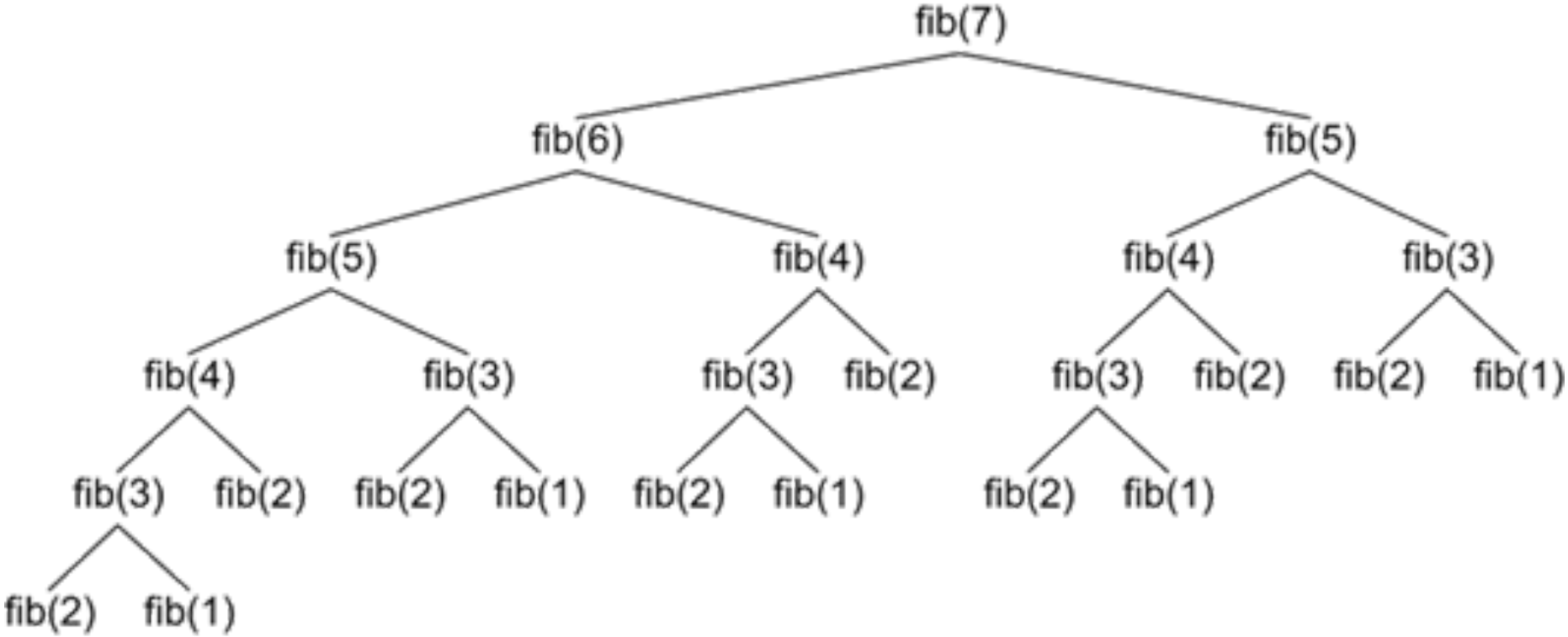
Calling Fib(7) Shows the Problem with Using Recursion



Calling Fib(7) Shows the Problem with Using Recursion



Calling Fib(7) Shows the Problem with Using Recursion

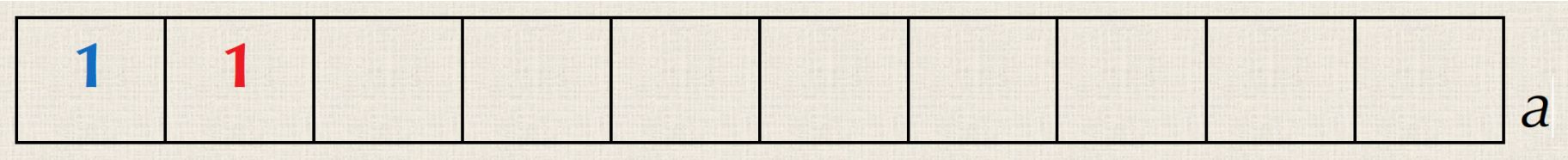


The Issue with Fibonacci Recursion

- Approximately how many calls do you think are made for RecFib(20)? What about RecFib(45)?
- When we call RecFib(n), there are ~ 2^n calls on the stack. For most values of n,
  - this will exhaust the memory allocated to the stack and produce what is called stack overflow, crashing the program.
- Key Point: We should evaluate whether recursion is a good approach for solving a problem based on whether we have many repeated calls with a chance of stack overflow

Computing Values "Bottom-Up" Avoids Many Recursive Calls

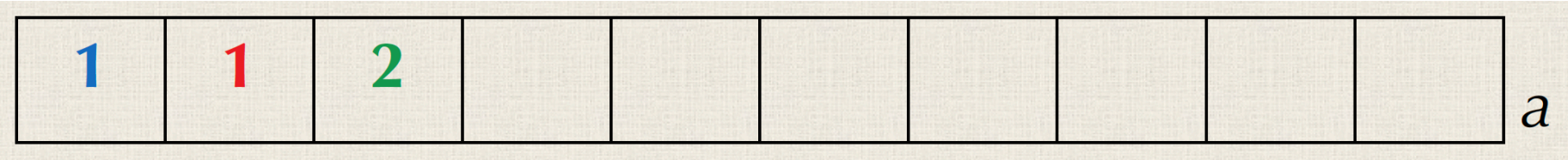
- Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.




```
Fibonacci(n)
        a ← array of length n
        a[0] ← 1
        a[1] ← 1
        for i ← 2 to n
                a[i] ← a[i-1] + a[i-2]
        return a
```

Computing Values "Bottom-Up" Avoids Many Recursive Calls

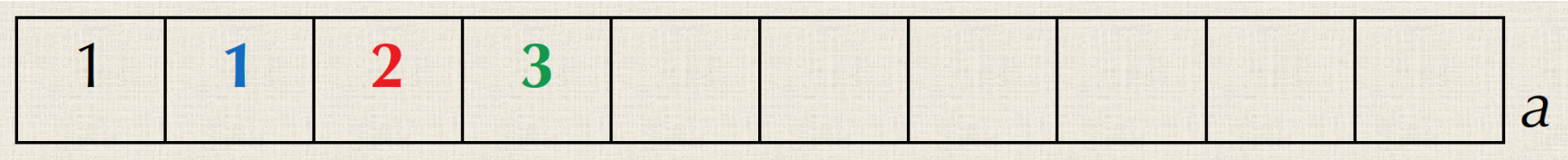
- Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.




```
Fibonacci(n)
        a ← array of length n
        a[0] ← 1
        a[1] ← 1
        for i ← 2 to n
                a[i] ← a[i-1] + a[i-2]
        return a
```

Computing Values "Bottom-Up" Avoids Many Recursive Calls

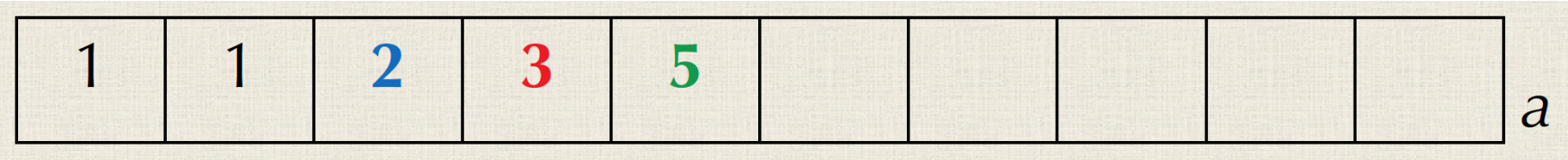
- Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.




```
Fibonacci(n)
        a ← array of length n
        a[0] ← 1
        a[1] ← 1
        for i ← 2 to n
                a[i] ← a[i-1] + a[i-2]
        return a
```

Computing Values "Bottom-Up" Avoids Many Recursive Calls

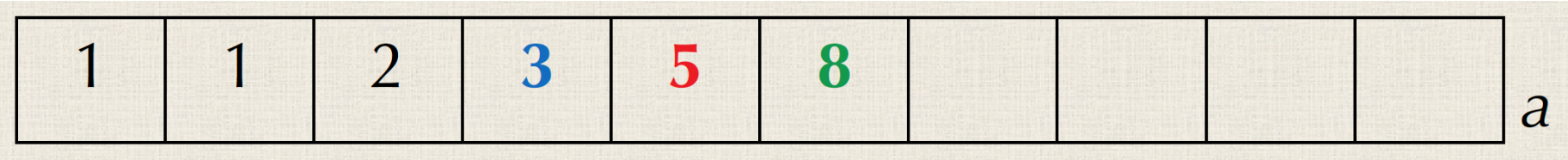
- Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.




```
Fibonacci(n)
        a ← array of length n
        a[0] ← 1
        a[1] ← 1
        for i ← 2 to n
                a[i] ← a[i-1] + a[i-2]
        return a
```

Computing Values "Bottom-Up" Avoids Many Recursive Calls

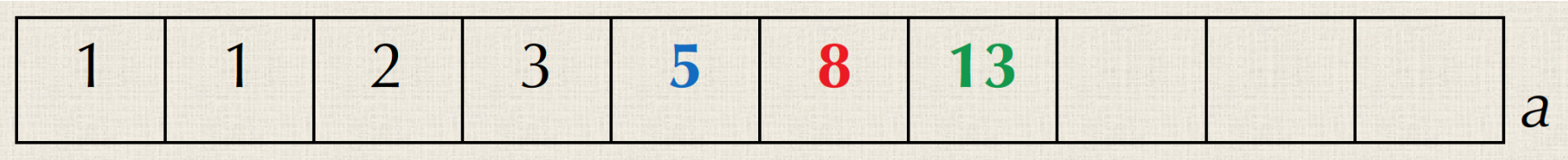
- Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.




```
Fibonacci(n)
        a ← array of length n
        a[0] ← 1
        a[1] ← 1
        for i ← 2 to n
                a[i] ← a[i-1] + a[i-2]
        return a
```

Computing Values "Bottom-Up" Avoids Many Recursive Calls

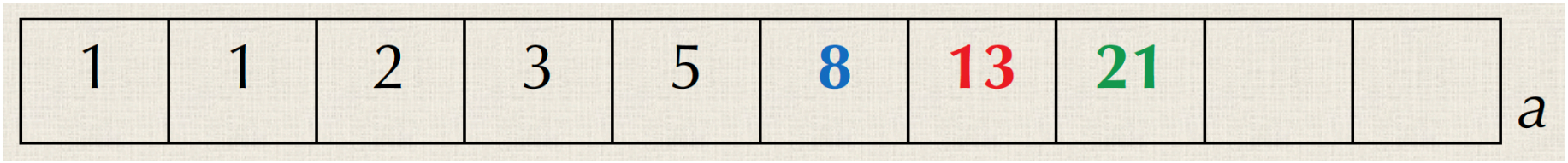
- Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.




```
Fibonacci(n)
        a ← array of length n
        a[0] ← 1
        a[1] ← 1
        for i ← 2 to n
                a[i] ← a[i-1] + a[i-2]
        return a
```

Computing Values "Bottom-Up" Avoids Many Recursive Calls

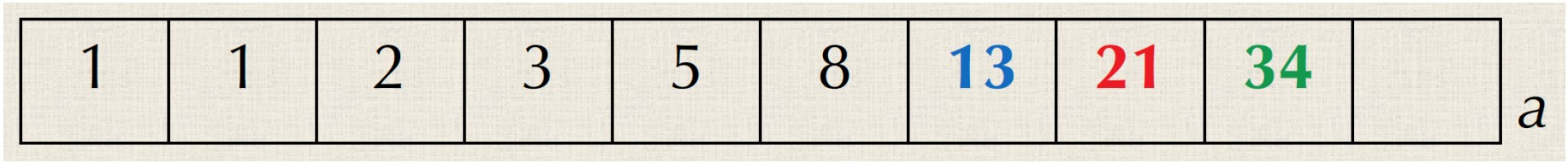
- Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.




```
Fibonacci(n)
        a ← array of length n
        a[0] ← 1
        a[1] ← 1
        for i ← 2 to n
                a[i] ← a[i-1] + a[i-2]
        return a
```

Computing Values "Bottom-Up" Avoids Many Recursive Calls

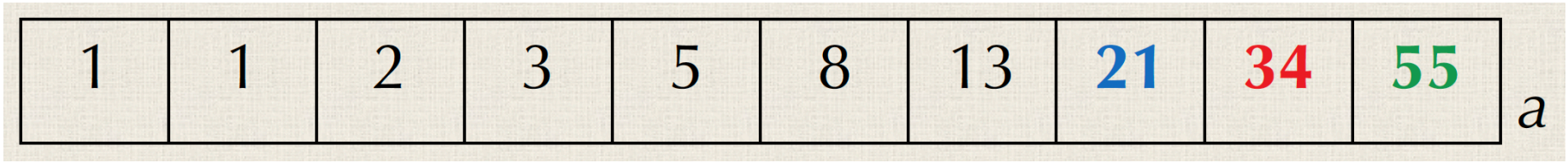
- Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.




```
Fibonacci(n)
        a ← array of length n
        a[0] ← 1
        a[1] ← 1
        for i ← 2 to n
                a[i] ← a[i-1] + a[i-2]
        return a
```

Computing Values "Bottom-Up" Avoids Many Recursive Calls

- Instead of computing Fibonacci numbers top-down recursively, we compute them bottom-up.



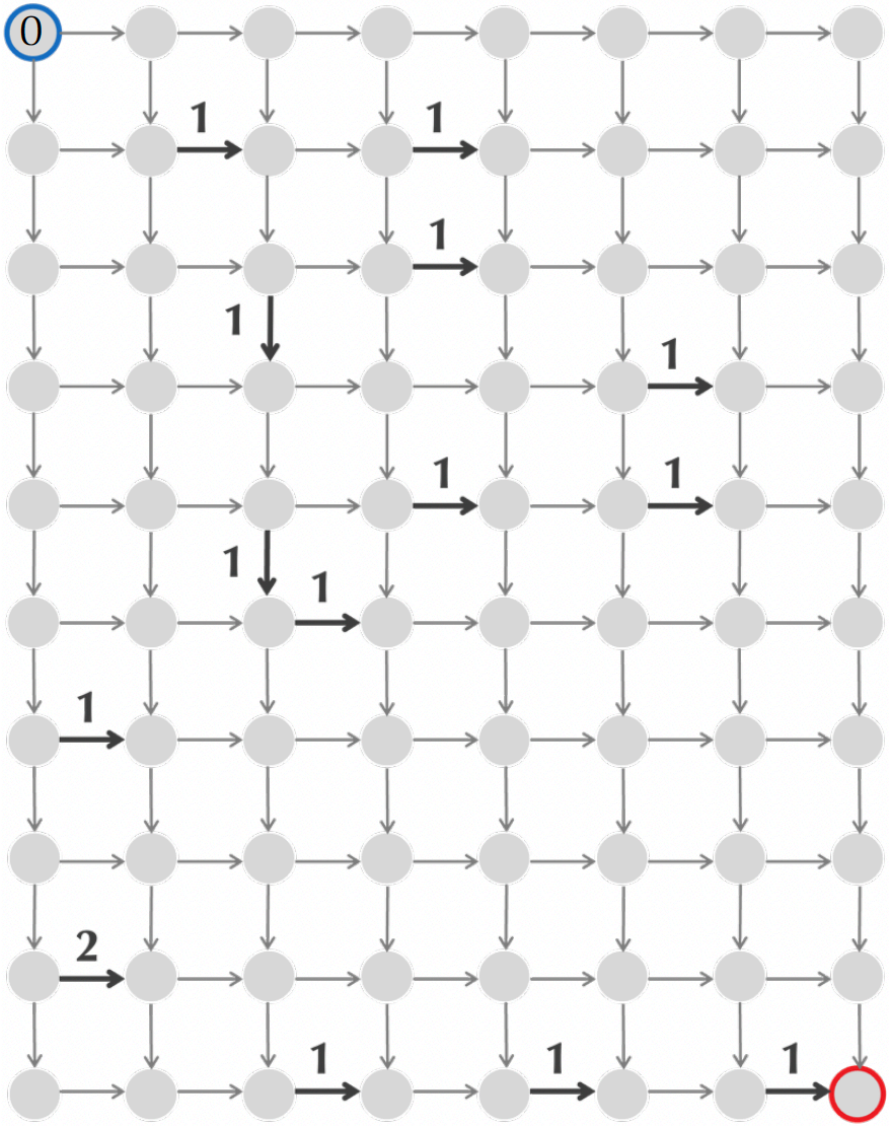

```
Fibonacci(n)
        a ← array of length n
        a[0] ← 1
        a[1] ← 1
        for i ← 2 to n
                a[i] ← a[i-1] + a[i-2]
        return a
```

Dynamic Programming

- Computing a recurrence relation bottom-up using an array is called dynamic programming.
- It is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming.
- The idea is to simply store the results of subproblems so that we do not have to re-compute them when needed later. This simple optimization typically reduces time complexities from exponential to polynomial.
- We can now use dynamic programming to find the best Manhattan tourist path

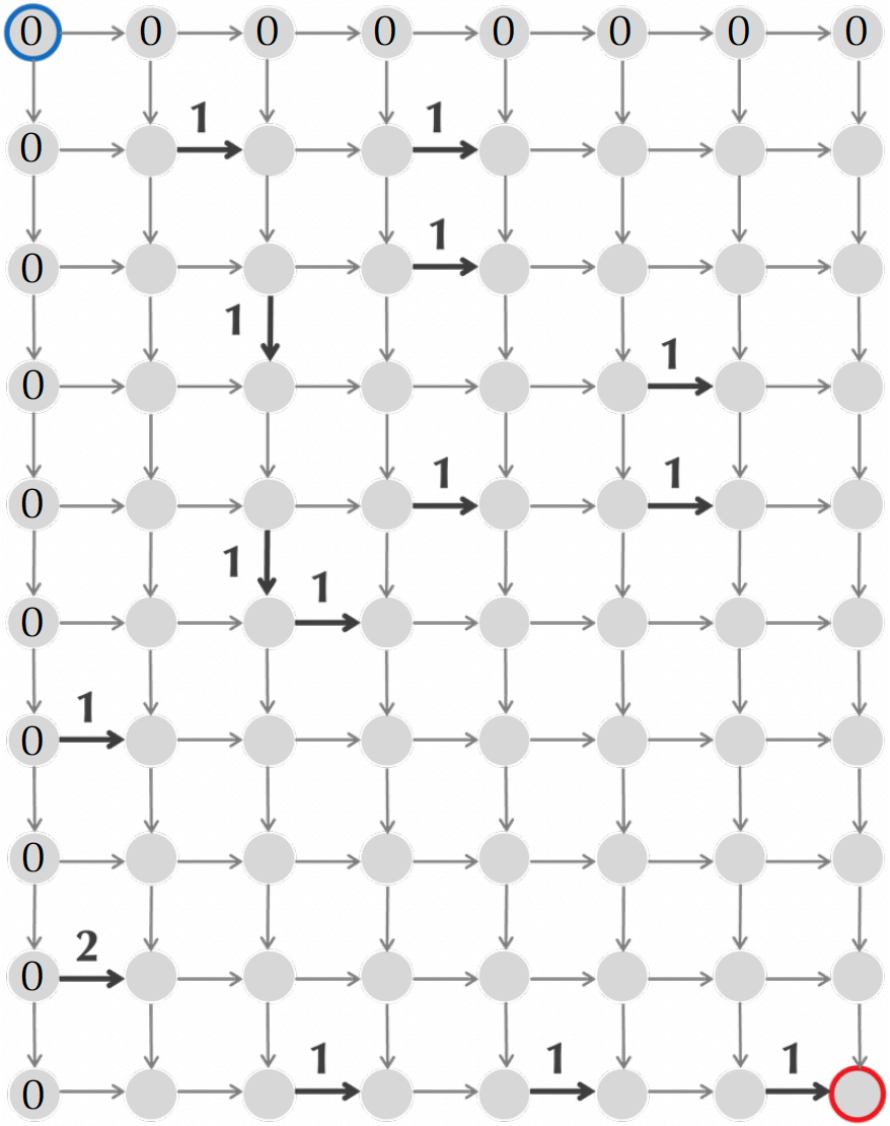
Dynamic Programming to find the best Manhattan tourist path

- Define s(i, j) as the score of a maximum weight path from the source to node (i, j).
- We start with s(0, 0), which must be equal to zero.
- Which other values of s(i, j) are easy to set?



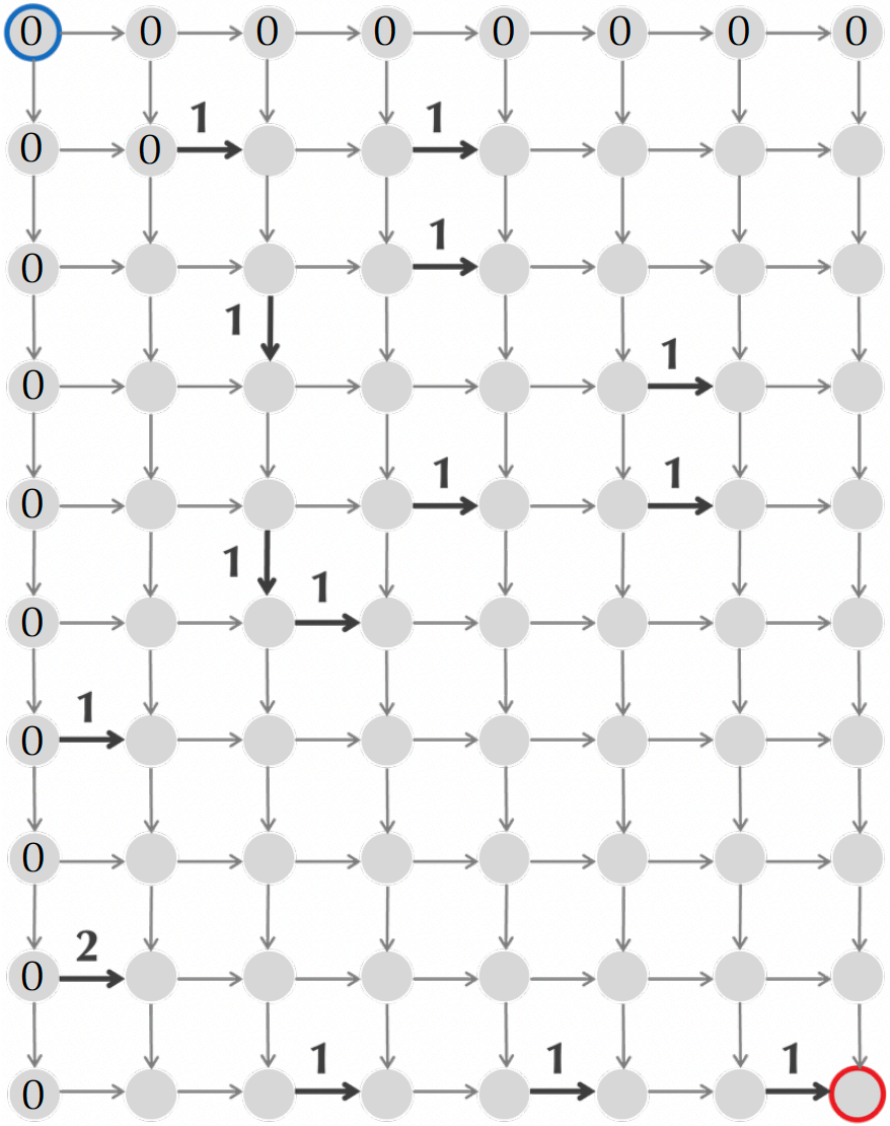
Dynamic Programming to find the best Manhattan tourist path

- We can set the scores of the entire first row and column because there is only one path into these nodes.
- Which value(s) of s(i, j) can we set now? Why?



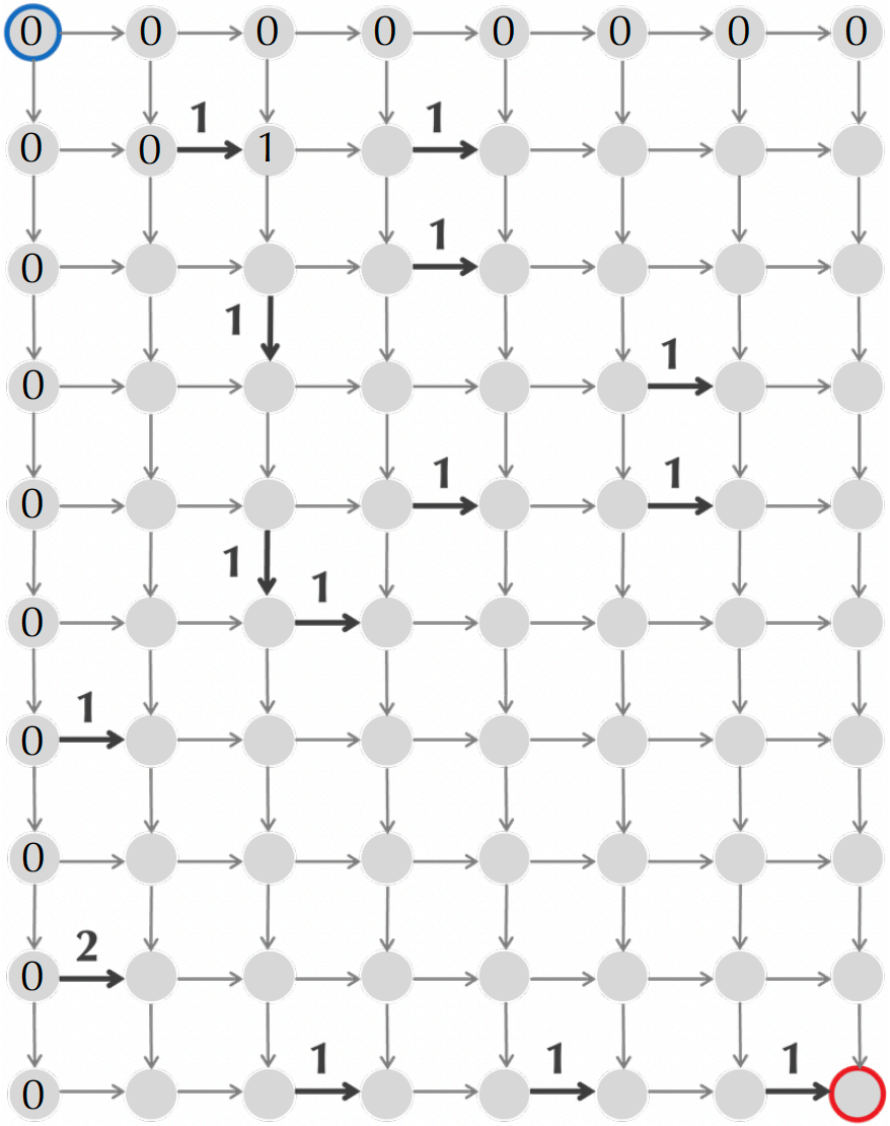
Dynamic Programming to find the best Manhattan tourist path

- We can only set s(1, 1) because we know that it must come from either the node above or the node to its left.



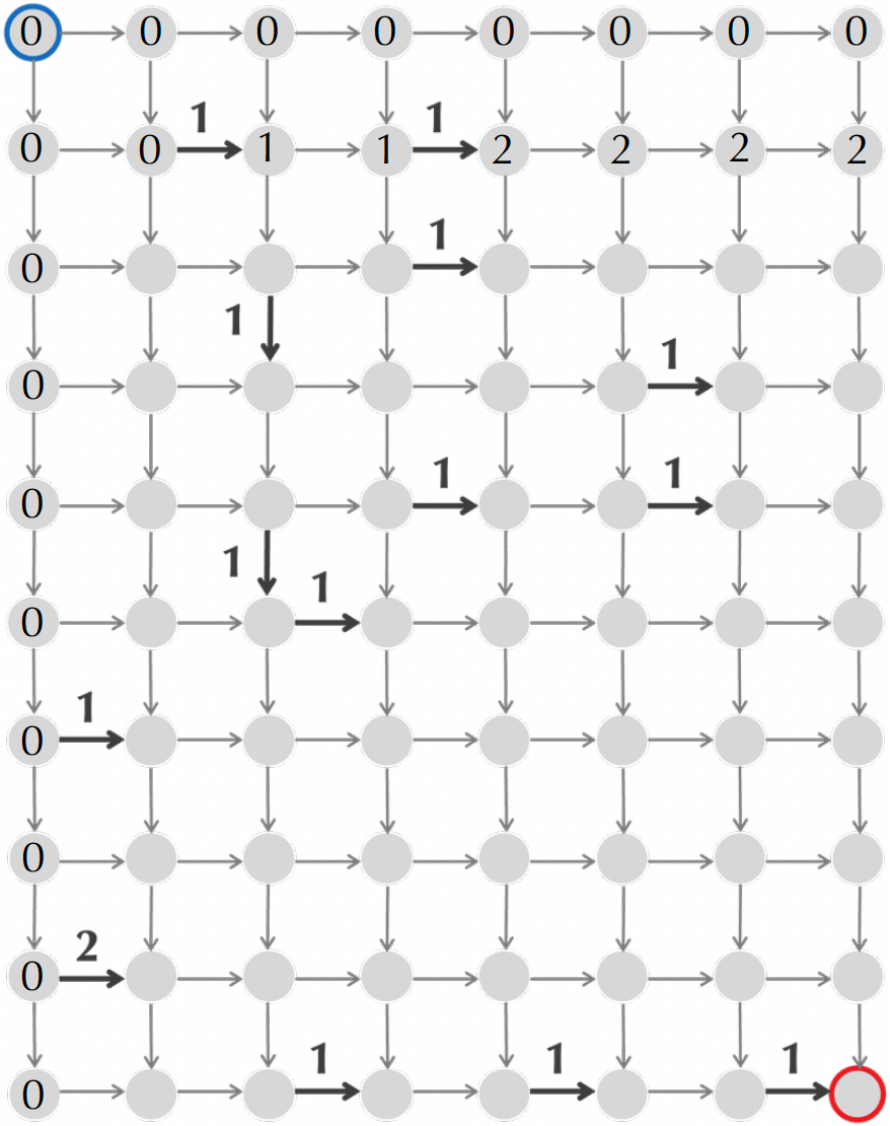
Dynamic Programming to find the best Manhattan tourist path

- We can now set s(1, 2) because the best path into (1, 2) must come from (0, 2) or (1, 2):
- s(1, 2) = s(1, 1) + 1 = 1.



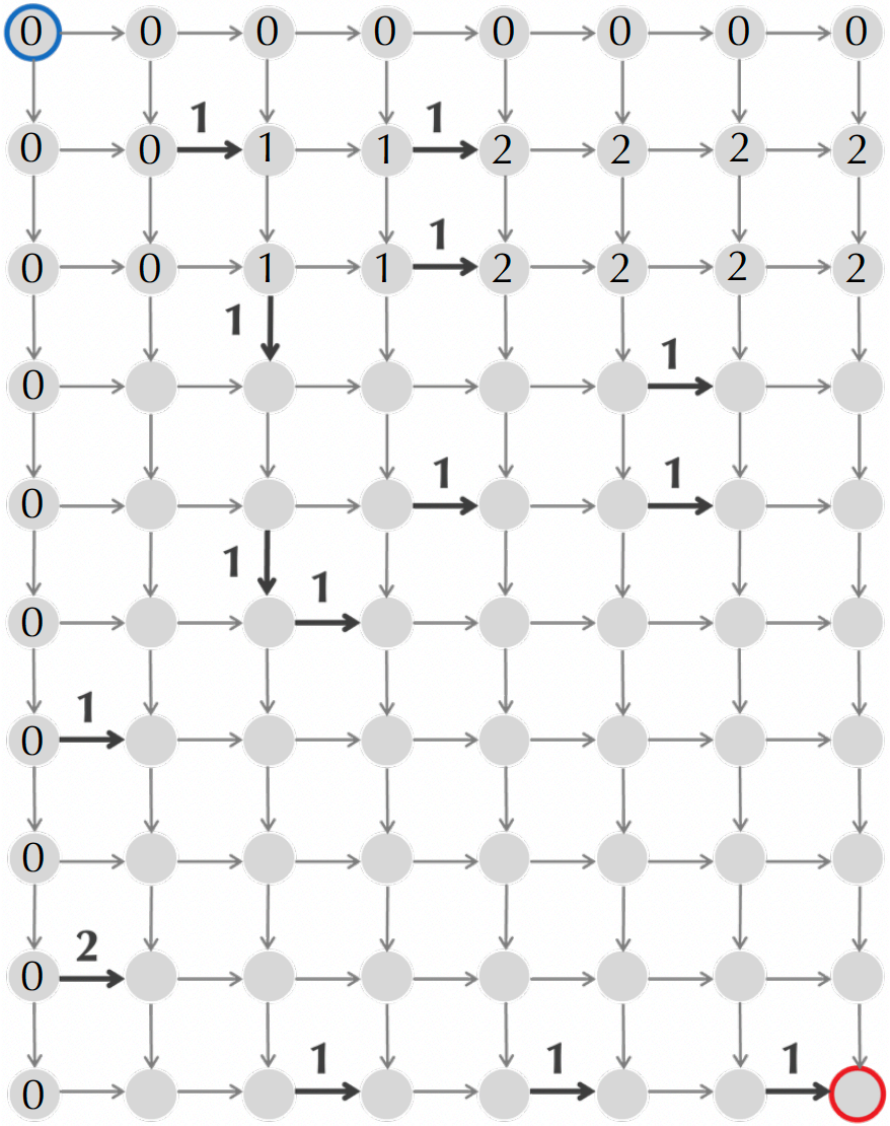
Dynamic Programming to find the best Manhattan tourist path

- Fill in the rest of row 1.



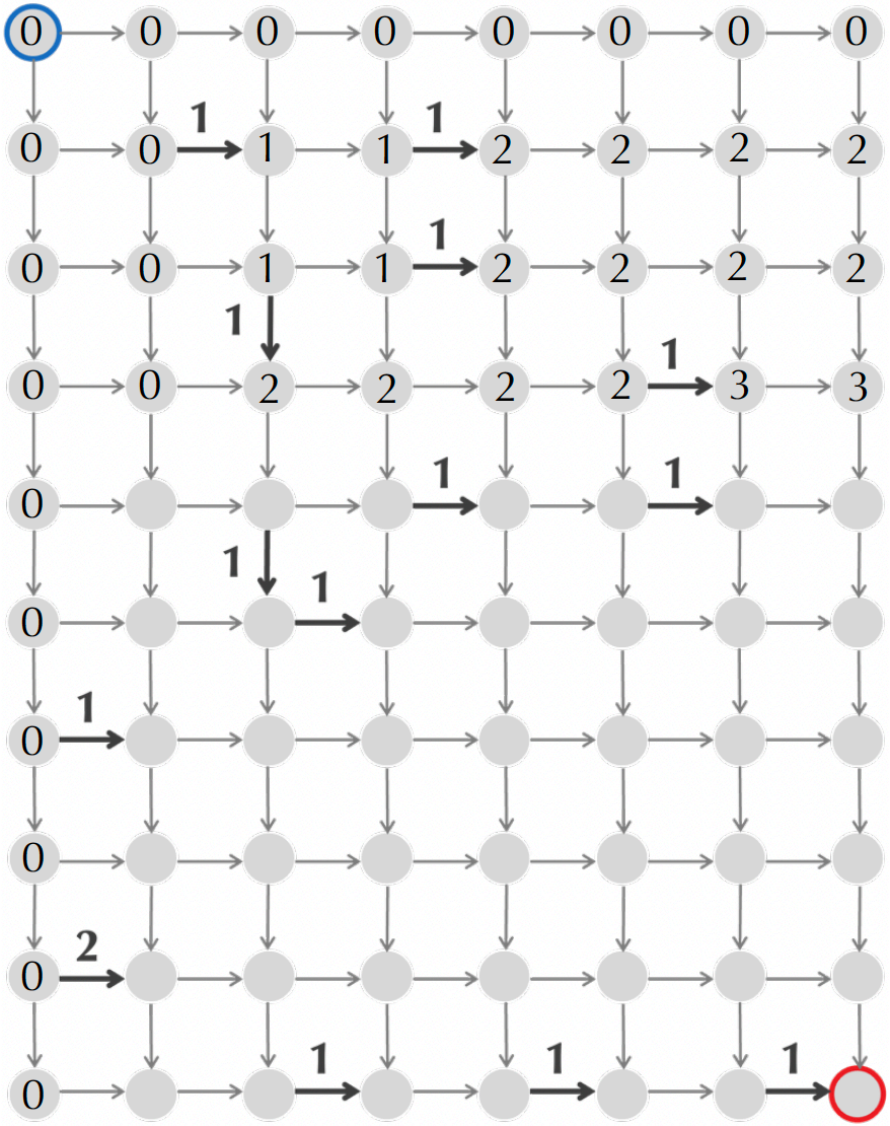
Dynamic Programming to find the best Manhattan tourist path

- In this way, we can fill in values of s(i, j) row by row.



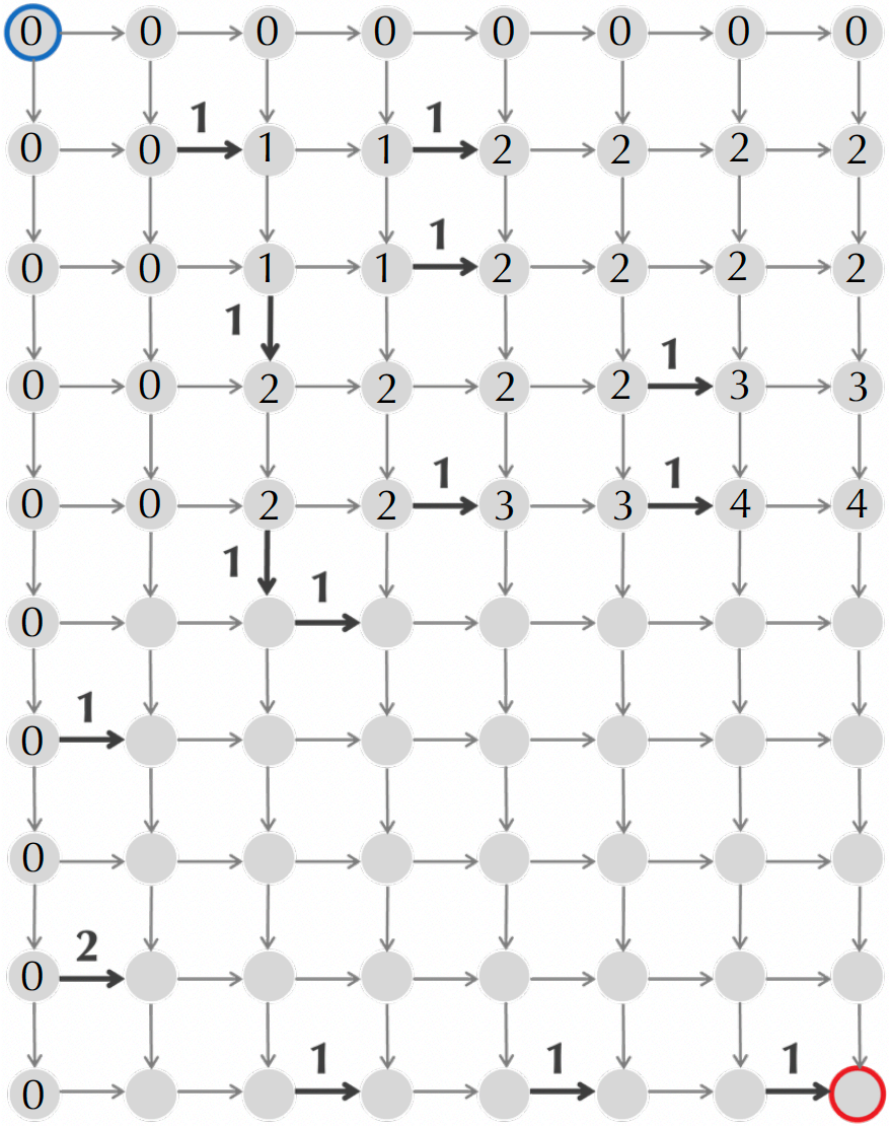
Dynamic Programming to find the best Manhattan tourist path

- In this way, we can fill in values of s(i, j) row by row.



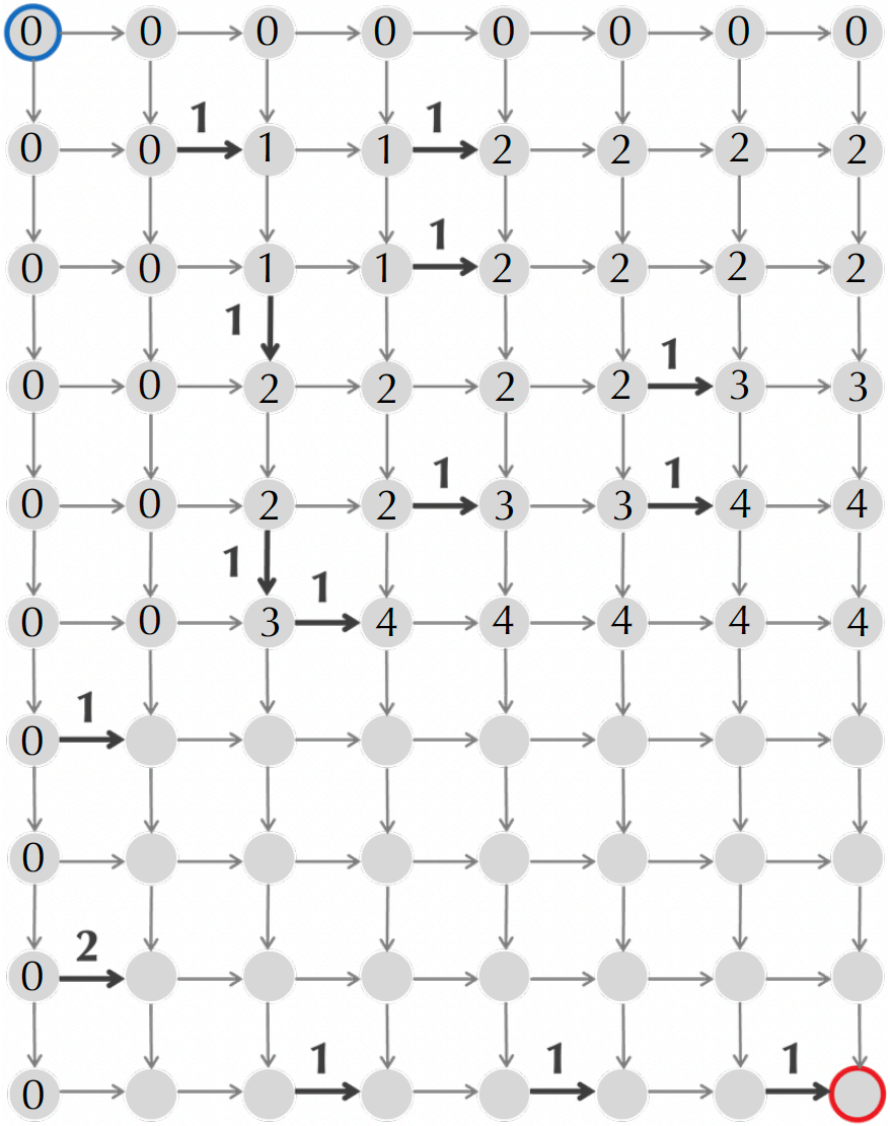
Dynamic Programming to find the best Manhattan tourist path

- In this way, we can fill in values of s(i, j) row by row.



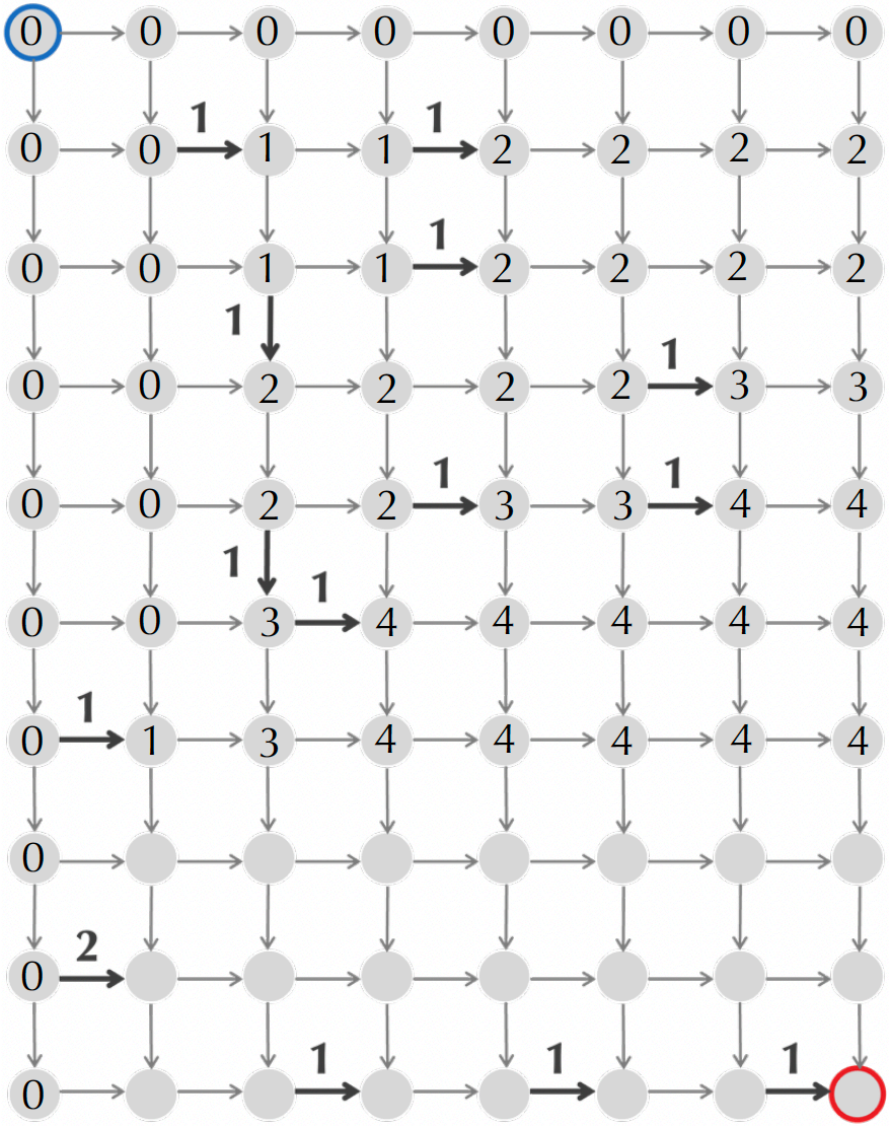
Dynamic Programming to find the best Manhattan tourist path

- In this way, we can fill in values of s(i, j) row by row.



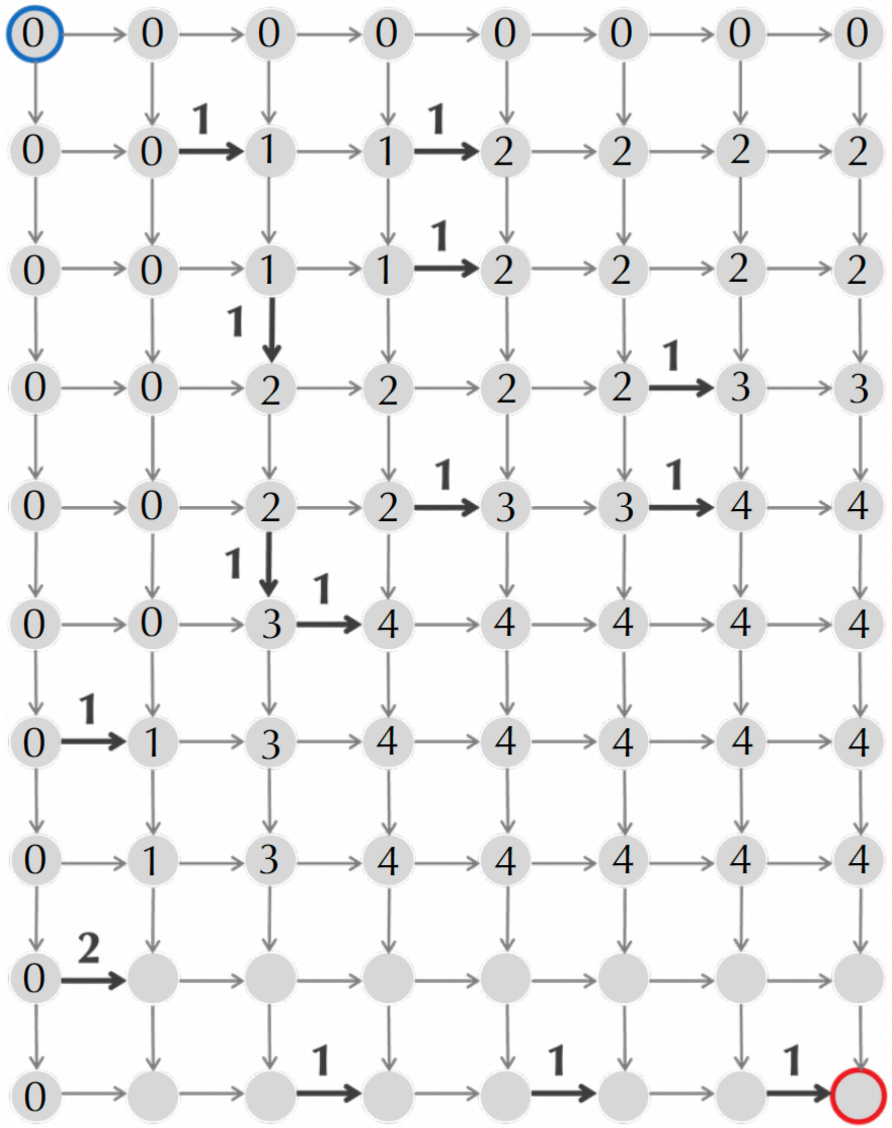
Dynamic Programming to find the best Manhattan tourist path

- In this way, we can fill in values of s(i, j) row by row.



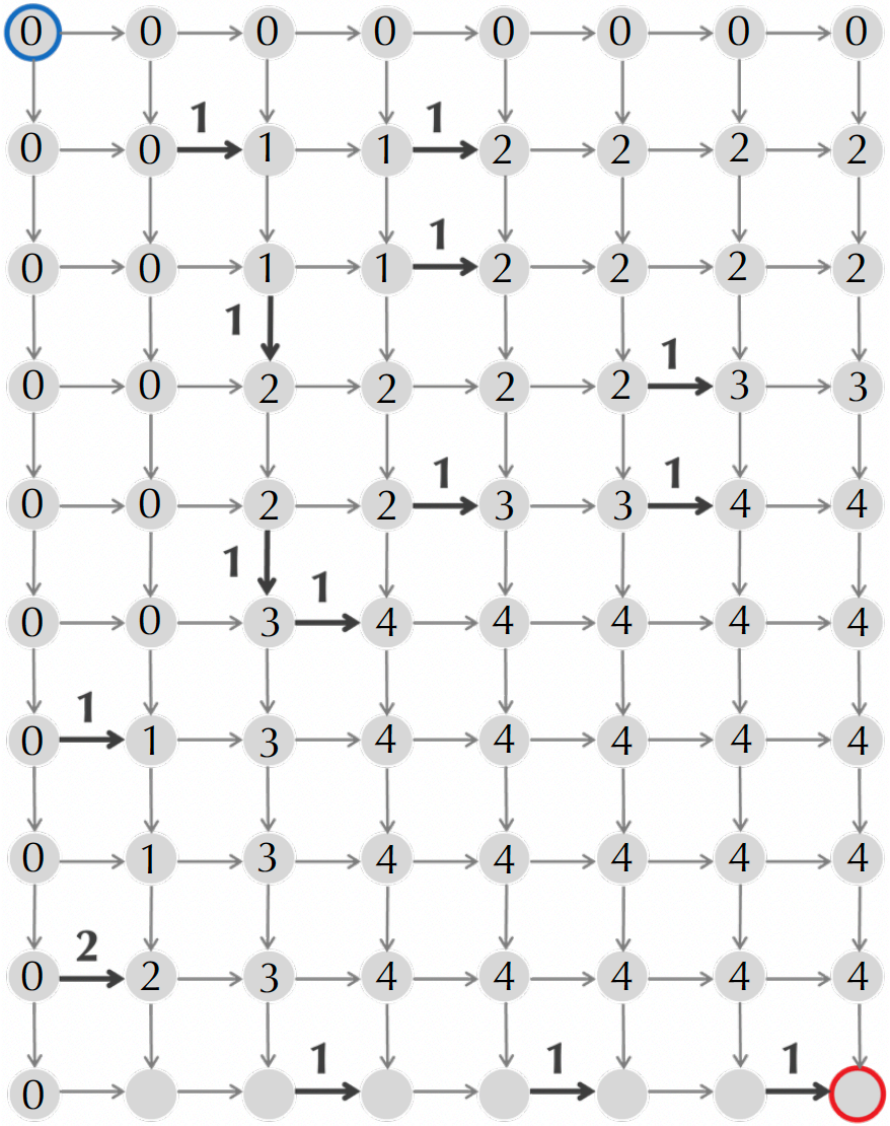
Dynamic Programming to find the best Manhattan tourist path

- In this way, we can fill in values of s(i, j) row by row.



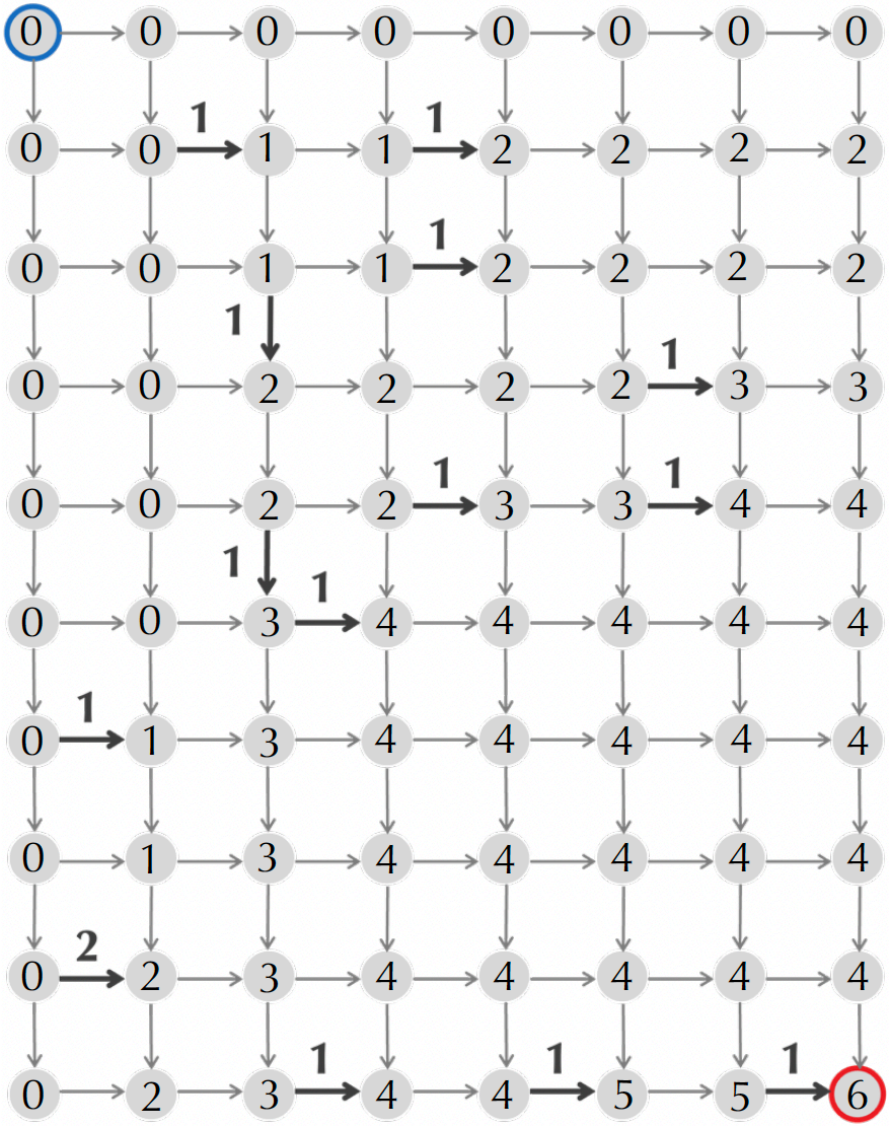
Dynamic Programming to find the best Manhattan tourist path

- In this way, we can fill in values of s(i, j) row by row.



Dynamic Programming to find the best Manhattan tourist path

- We now have a concrete dynamic programming algorithm that will find the longest path in any rectangular grid!



Returning to Manhattan

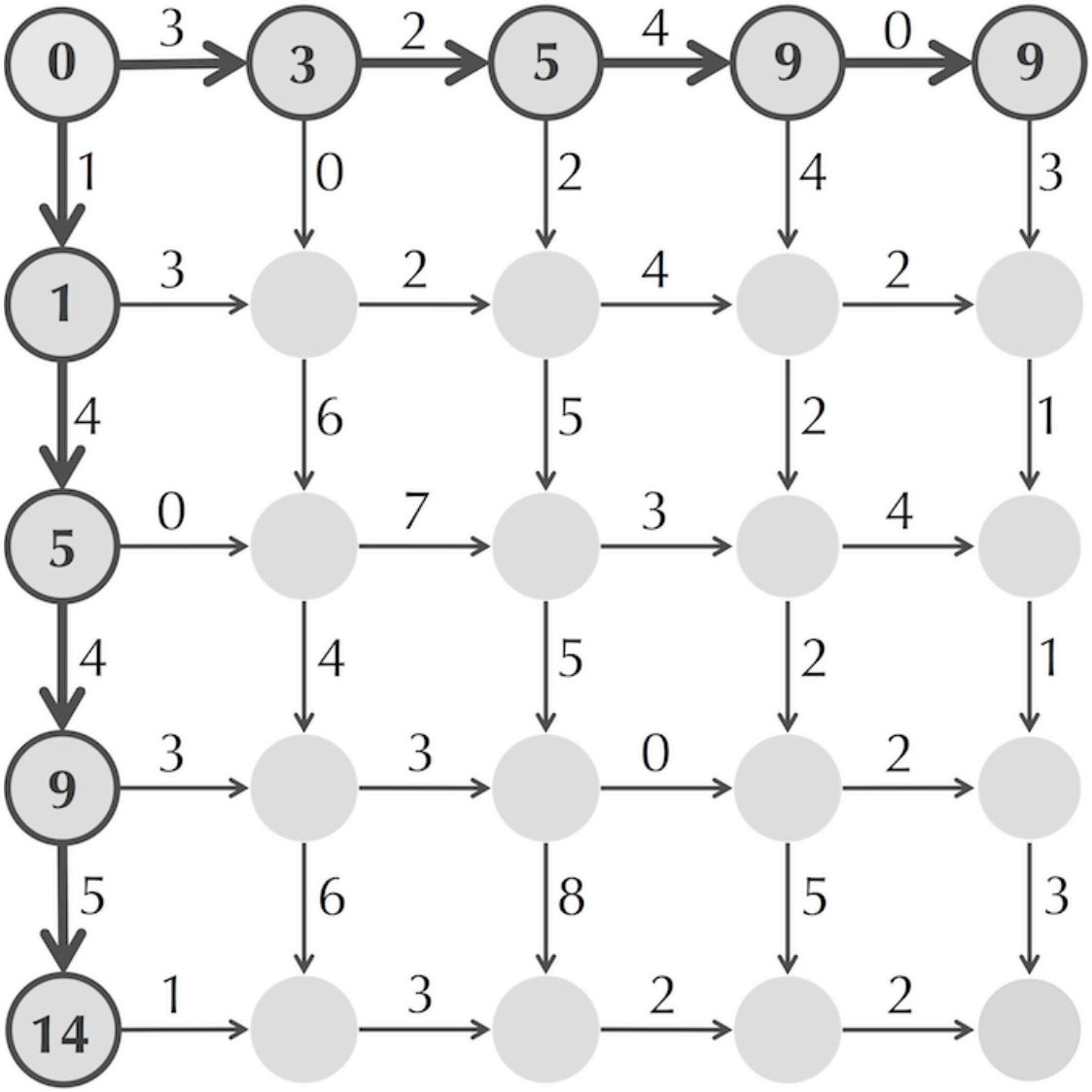
- Manhattan Tourist Problem:
  - Input: A weighted n x m rectangular grid (n + 1 rows and m + 1 columns).
  - Output: A longest path from source (0, 0) to sink (n, m) in the grid.
- Exercise: Find a recurrence relation for the length of a longest path from (0,0) to node (i, j), which we will call length(i,j).
- Answer: length(i,j) = max{ length(i – 1,j) + weight(vertical edge into i,j), length(i, j – 1) + weight(horizontal edge into i,j)}.

Returning to Manhattan

- STOP: Will a recursive algorithm for Manhattan Tourist have the same problem that the recursive change-making function encountered?
- Answer: Yes! Because the same length(i, j) can get re-computed many times…

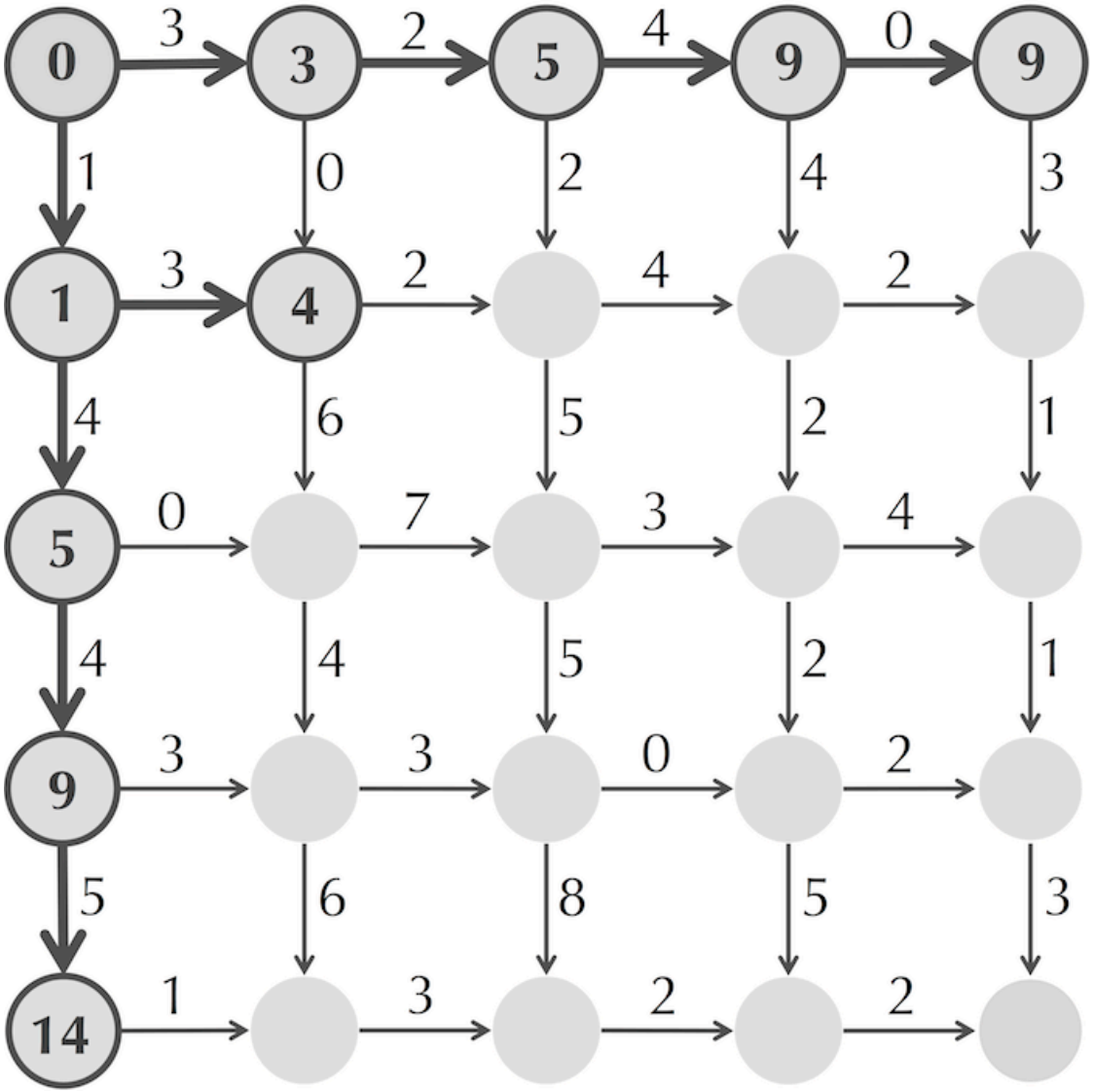
Let's Use Dynamic Programming Instead

- Which element of the table should we fill in next and what should its value be?



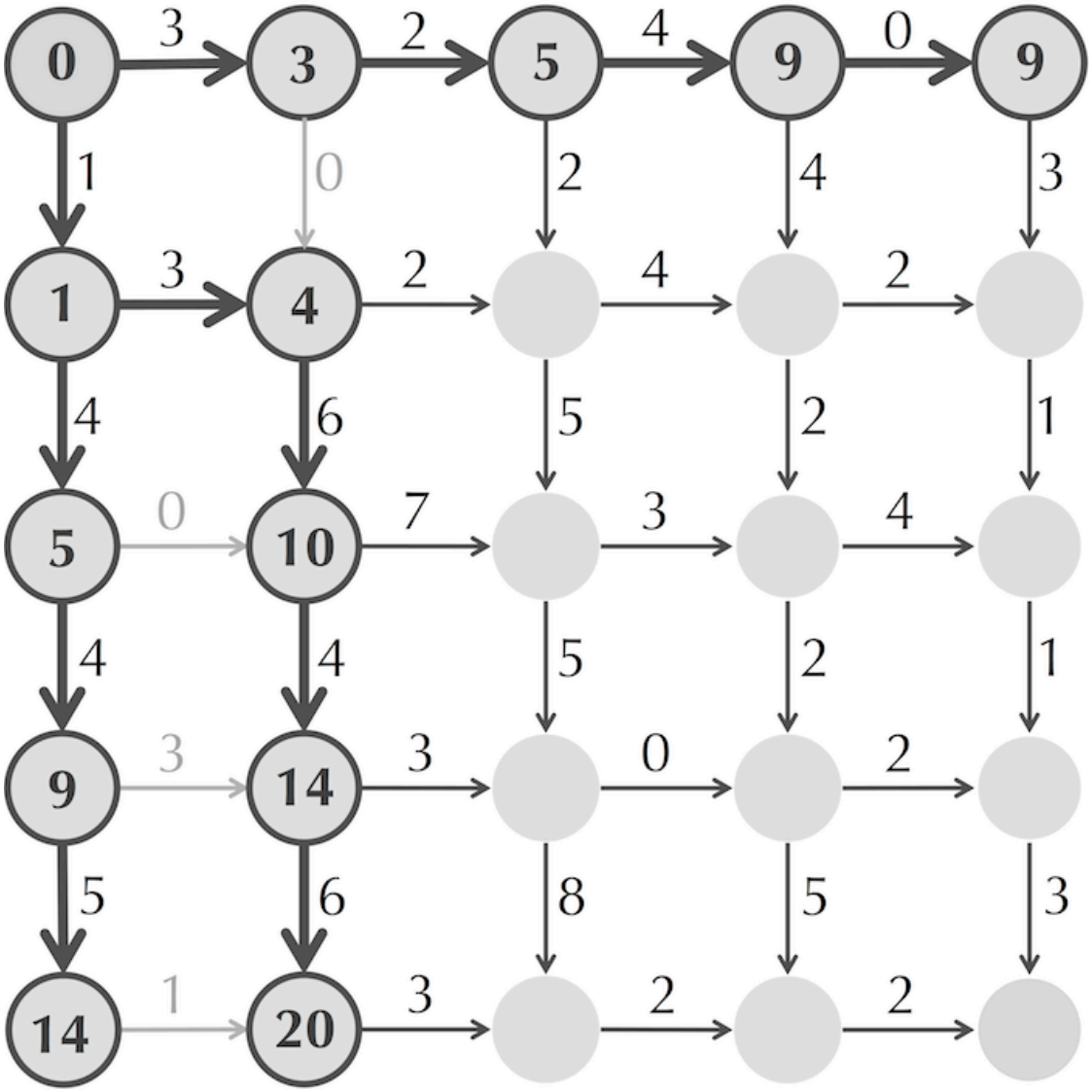
Let's Use Dynamic Programming Instead

- Answer: We only know the values of MaxWeight for the two nodes adjacent to the node (1, 1); it gets the value max(3+0, 1+3) = 4.
- STOP: Which elements should we fill in next and what should their values be?



Let's Use Dynamic Programming Instead

- Answer: We can fill in all of row 1 or all of column 1 (it doesn't matter which).
- Exercise: Fill in the remaining values of length for this network.



Let's Use Dynamic Programming Instead

- STOP: Now do you see a longest path in this grid? How might we find one in general?
- Now we can look into the Sequence Alignment problem in a different light!