CodeIT

Tabulation

Tabulation

- With **memoization**, the procedure was to recurse into sub problems, and build back up, caching results when possible

- This is known as a **top down approach**

- With tabulation, we simply start at the **smallest sub problem** (the base case), and work our way up in a bottom up approach

Problem #1: Fibonacci (Tabulation)

fib(7)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

fib(7)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

0	1	2	3	4	5	6	7	8

fib(7)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

1	1							
0	1	2	3	4	5	6	7	8

fib(7)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

1	1							
0	1	2	3	4	5	6	7	8

fib(7)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

1	1	2						
0	1	2	3	4	5	6	7	8

fib(7)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

1	1	2	3					
0	1	2	3	4	5	6	7	8

fib(7)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

1	1	2	3	5				
0	1	2	3	4	5	6	7	8

fib(7)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

1	1	2	3	5	8			
0	1	2	3	4	5	6	7	8

fib(7)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

1	1	2	3	5	8	13		
0	1	2	3	4	5	6	7	8

fib(7)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

1	1	2	3	5	8	13	21	
0	1	2	3	4	5	6	7	8

fib(7)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

1	1	2	3	5	8	13	21	34
0	1	2	3	4	5	6	7	8

fib(7)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

1	1	2	3	5	8	13	21	34
0	1	2	3	4	5	6	7	8

time complexity: n
space complexity: n

fibTab

```python
def fibTab(n):
    table = [None] * (n + 1)

    table[0] = 1
    table[1] = 1

    for i in range(2, n + 1):
        table[i] = table[i - 1] + table[i - 2]

    return table[n]
```

Problem #2: Unique Paths (Tabulation)

Given an m x n grid, assuming you can only move **right & down**, calculate the total number of unique paths to get from **top left to bottom right**

Problem #2: Unique Paths (Tabulation)

Given an m x n grid, assuming you can only move **right & down**, calculate the total number of unique paths to get from **top left to bottom right**

Note: This is an example of a **2D DP problem**. We have 2 variables that affect our cache result (m & n). As such, we have to initialise a 2D table!

uniquePaths (4, 3)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

uniquePaths (4, 3)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

m

n

	0	1	2	3	4
0					
1					
2					
3					

uniquePaths (4, 3)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

m

n

	0	1	2	3	4
0	0	0	0	0	0
1	0	1			
2	0				
3	0				

uniquePaths (4, 3)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

m

n

	0	1	2	3	4
0	0	0	0	0	0
1	0	1			
2	0				
3	0				

uniquePaths (4, 3)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

m

n

	0	1	2	3	4
0	0	0	0	0	0
1	0	1			
2	0				
3	0				

Recall Optimal Substructure:

uniquePaths(m, n) =

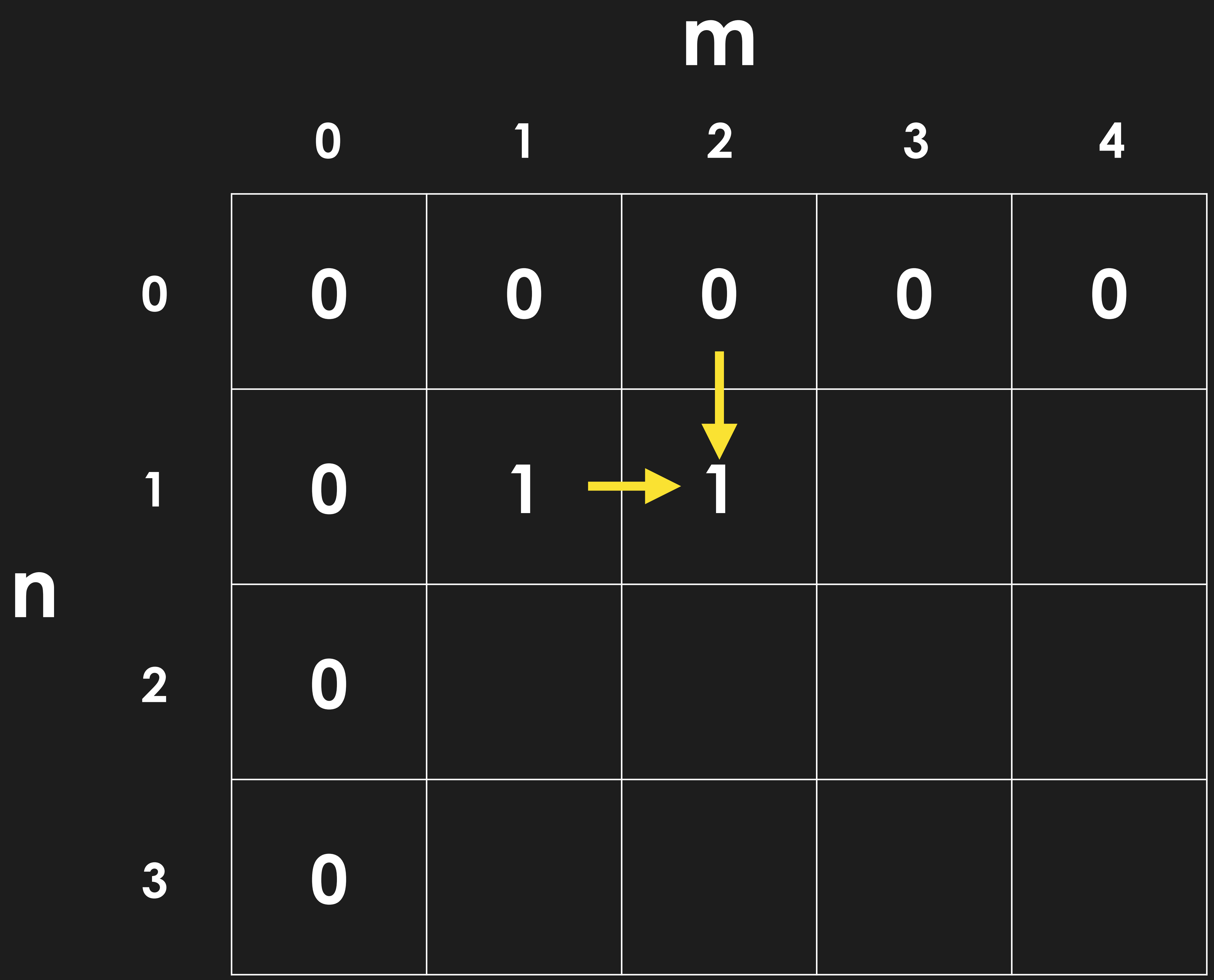
uniquePaths(m - 1, n) +

uniquePaths(n - 1, m)

uniquePaths (4, 3)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate



m

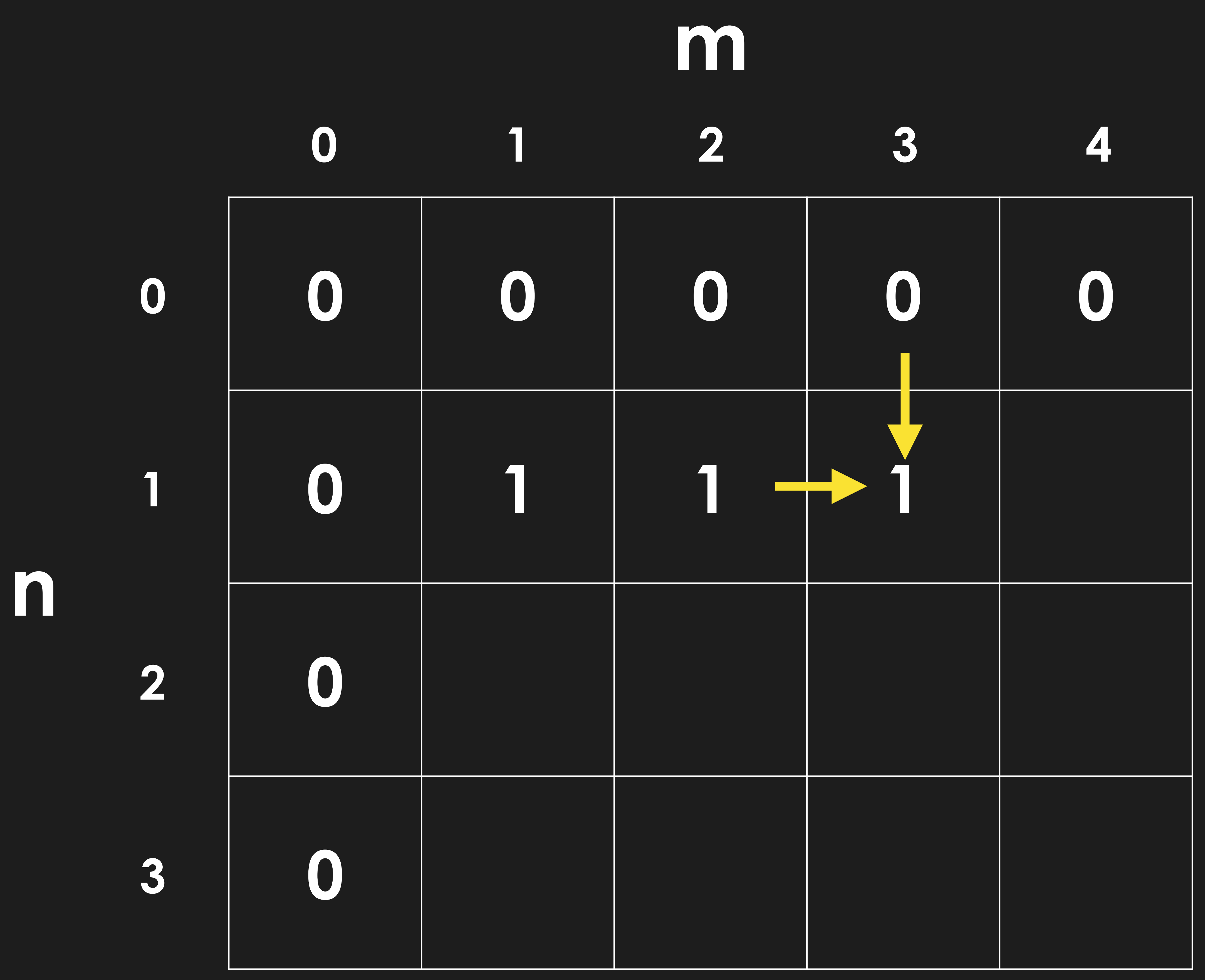
n

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1		
2	0				
3	0				

uniquePaths (4, 3)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

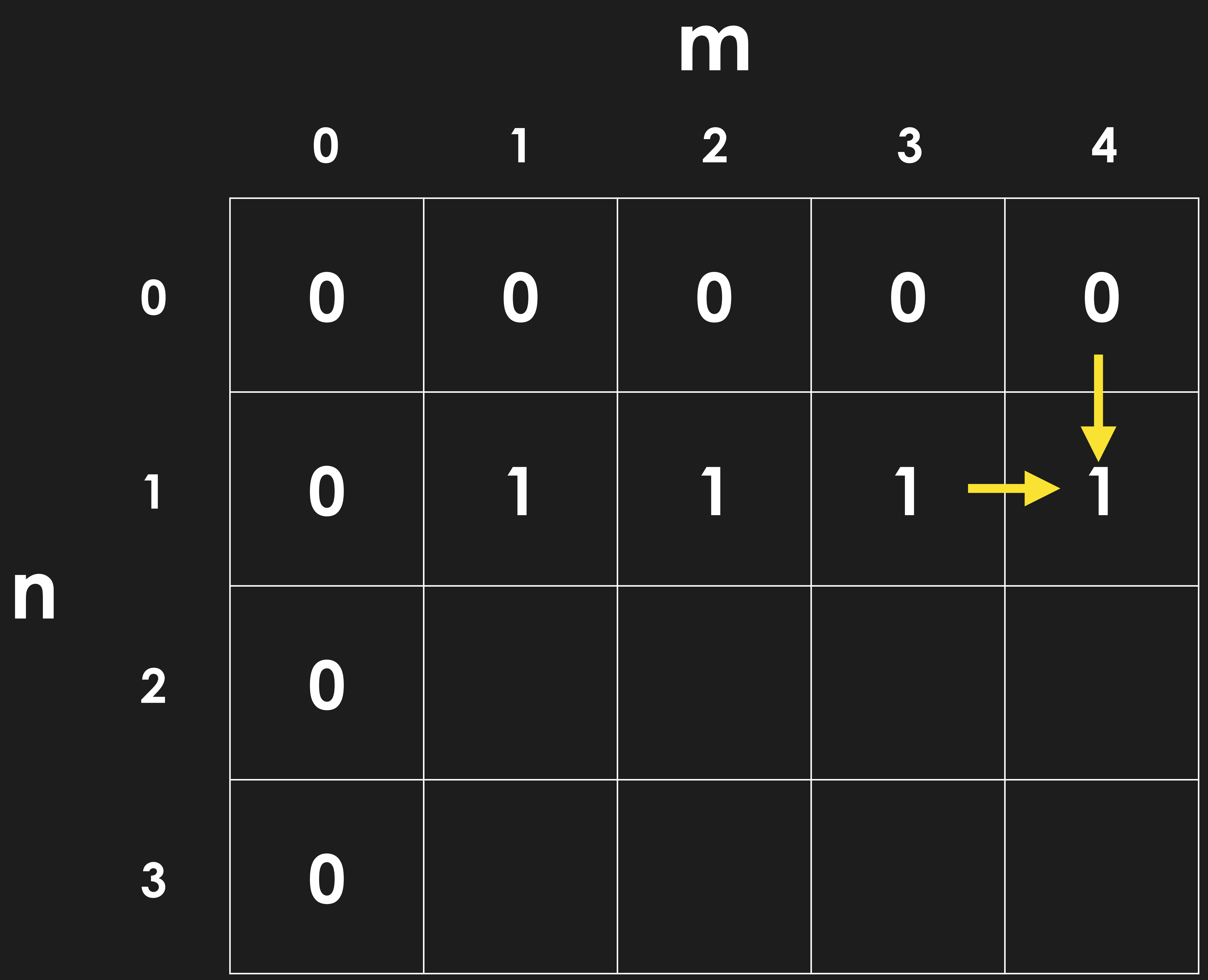


	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	
2	0				
3	0				

uniquePaths (4, 3)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate



m

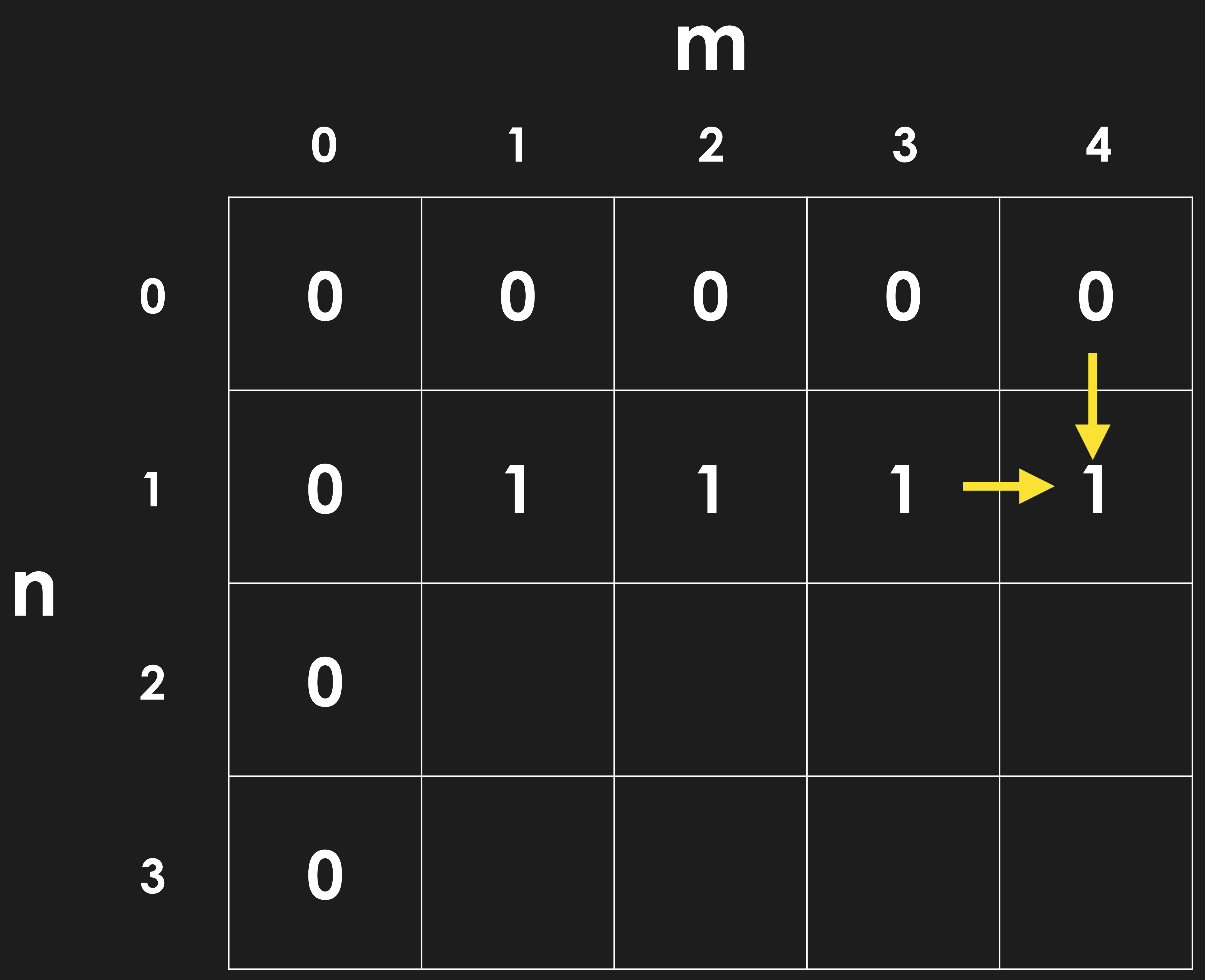
n

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0				
3	0				

uniquePaths (4, 3)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate



m

n

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0				
3	0				

uniquePaths (4, 3)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

m

n

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0	1	2	3	4
3	0				

uniquePaths (4, 3)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

m

n

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0	1	2	3	4
3	0	1	3	6	10

uniquePaths (4, 3)

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

m

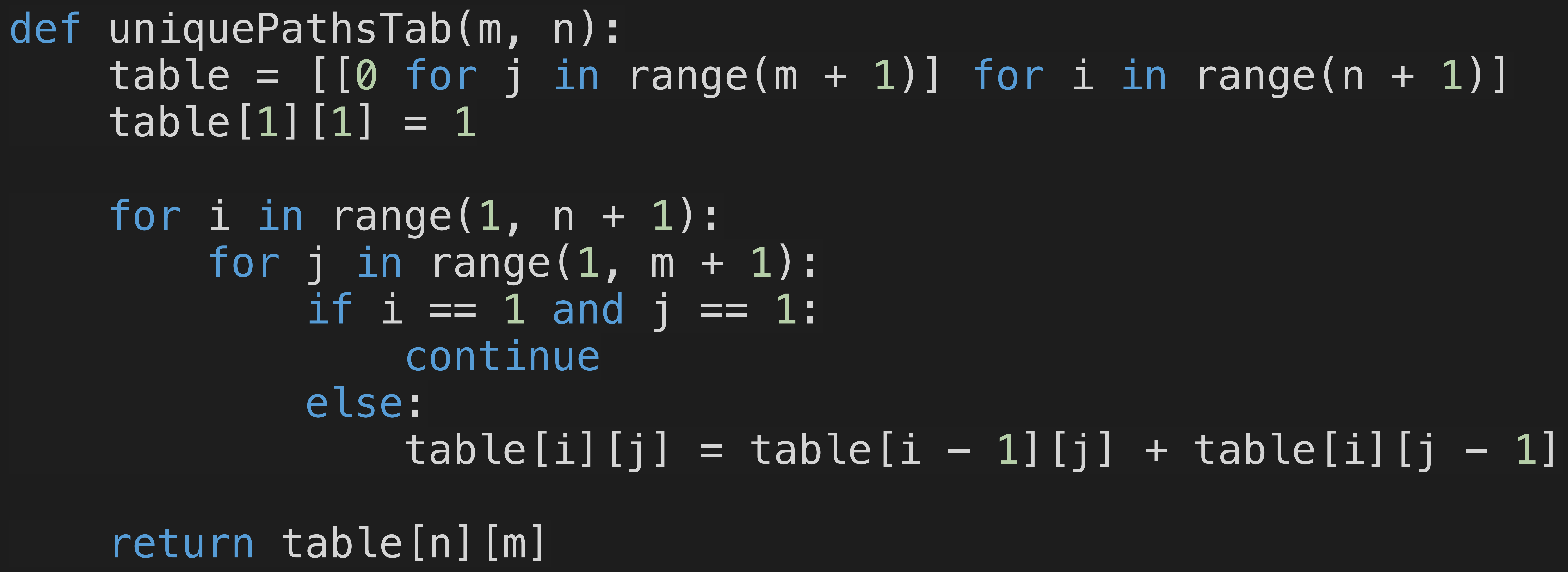
n

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0	1	2	3	4
3	0	1	3	6	10

time complexity: n * m

space complexity: n * m

uniquePathsTab



```python
def uniquePathsTab(m, n):
    table = [[0 for j in range(m + 1)] for i in range(n + 1)]
    table[1][1] = 1

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if i == 1 and j == 1:
                continue
            else:
                table[i][j] = table[i - 1][j] + table[i][j - 1]

    return table[n][m]
```

Problem #3: Can Sum (Tabulation)

Given an array of integers, and a target sum, print out whether there exists a combination of the array of numbers, with repetition, that adds to the target sum

Problem #3: Can Sum (Tabulation)

Given an array of integers, and a target sum, print out whether there exists a combination of the array of numbers, with repetition, that adds to the target sum

Initialising the table:

1. How many **variables** are being changed in each recursive call?
2. What are the **total number of "states"** that the changed variable can be in?

Problem #3: Can Sum (Tabulation)

Given an array of integers, and a target sum, print out whether there exists a combination of the array of numbers, with repetition, that adds to the target sum

Initialising the table:

1. How many **variables** are being changed in each recursive call? **1 (target)**
2. What are the **total number of "states"** that the changed variable can be in? **target**

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

0	1	2	3	4	5	6	7	8	9	10

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

T										
0	1	2	3	4	5	6	7	8	9	10

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

T										
0	1	2	3	4	5	6	7	8	9	10

```
For i in range(target):
    if i is True:
        for n in array:
            i + n = True
```

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

T										
0	1	2	3	4	5	6	7	8	9	10

```
For i in range(target):
    if i is True:
        for n in array:
            i + n = True
```

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

i										
T			T			T	T			
0	1	2	3	4	5	6	7	8	9	10

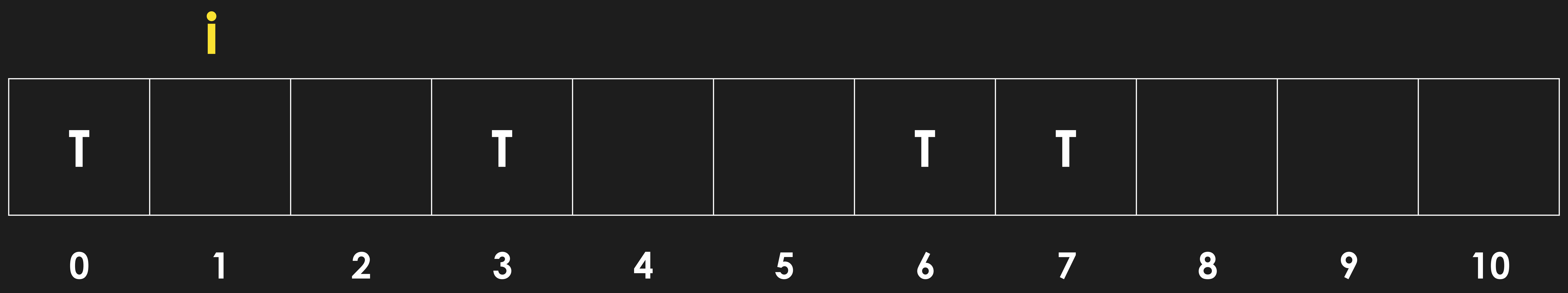
```
For i in range(target):
    if i is True:
        for n in array:
            i + n = True
```

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate



T			T			T	T			
0	1	2	3	4	5	6	7	8	9	10

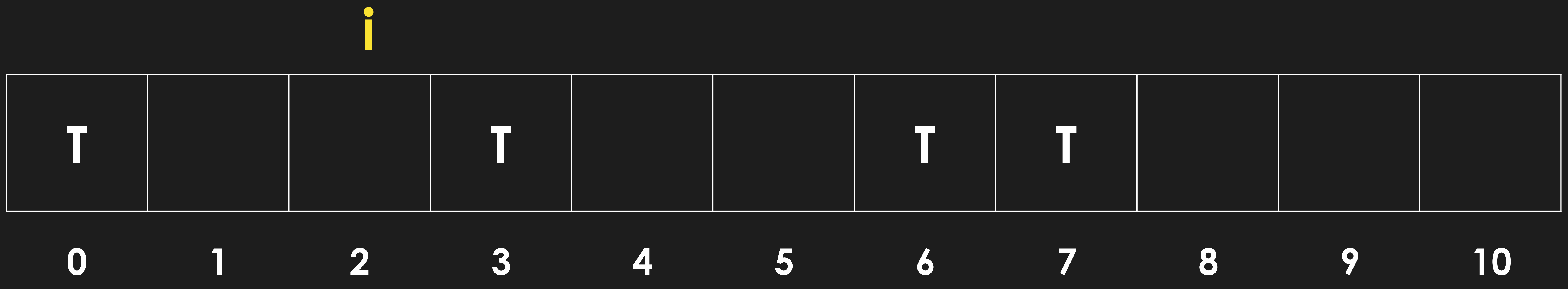
```
For i in range(target):
    if i is True:
        for n in array:
            i + n = True
```

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate



T			T			T	T			
0	1	2	3	4	5	6	7	8	9	10

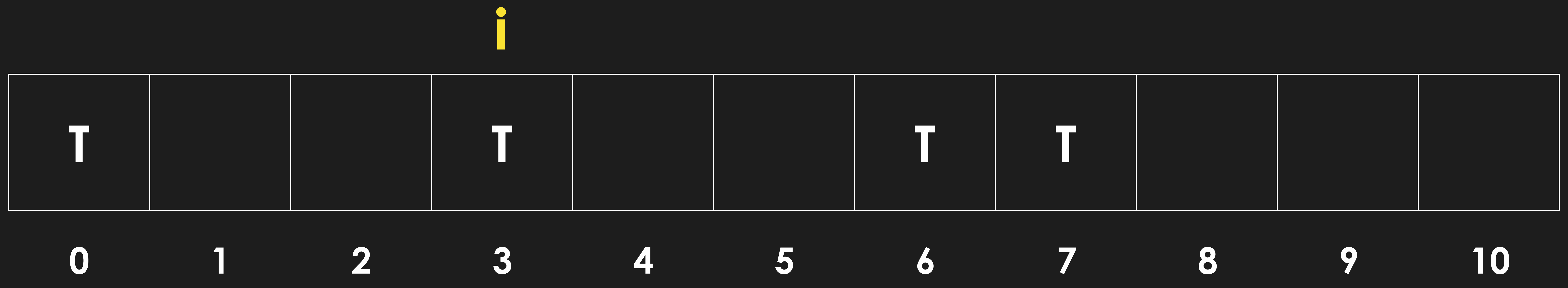
```
For i in range(target):
    if i is True:
        for n in array:
            i + n = True
```

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate



T			T			T	T			
0	1	2	3	4	5	6	7	8	9	10

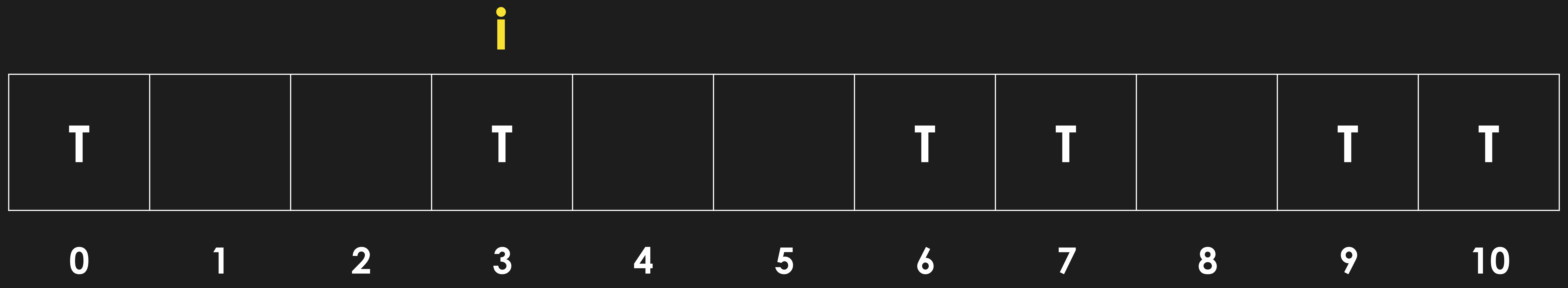
```
For i in range(target):
    if i is True:
        for n in array:
            i + n = True
```

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate



T			T			T	T		T	T
0	1	2	3	4	5	6	7	8	9	10

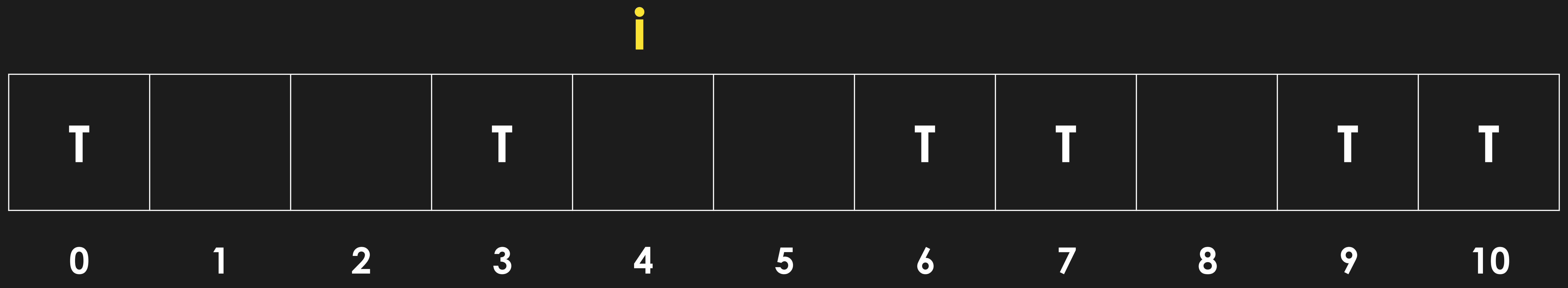
```
For i in range(target):
    if i is True:
        for n in array:
            i + n = True
```

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate



T			T			T	T		T	T
0	1	2	3	4	5	6	7	8	9	10

```
For i in range(target):
    if i is True:
        for n in array:
            i + n = True
```

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

					i					
T			T			T	T		T	T
0	1	2	3	4	5	6	7	8	9	10

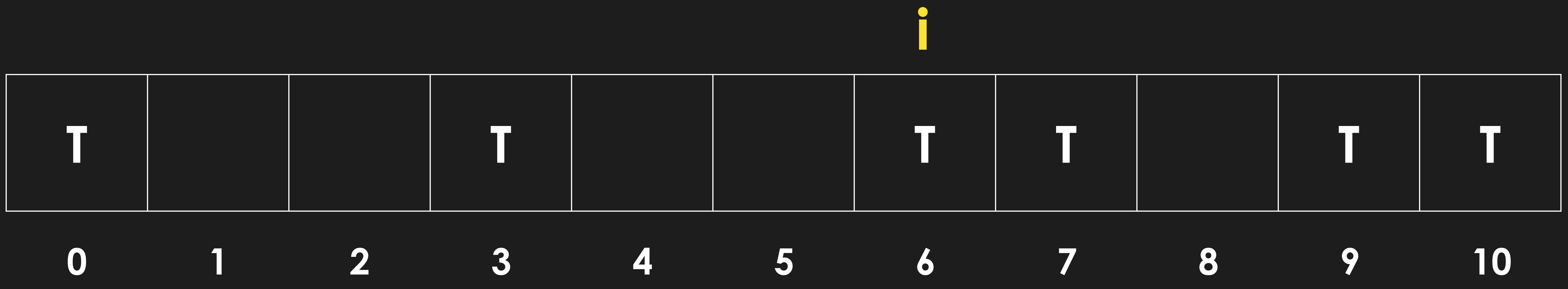
```
For i in range(target):
    if i is True:
        for n in array:
            i + n = True
```

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate



T			T			T	T		T	T
0	1	2	3	4	5	6	7	8	9	10

```
For i in range(target):
    if i is True:
        for n in array:
            i + n = True
```

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

							i			
T			T			T	T		T	T
0	1	2	3	4	5	6	7	8	9	10

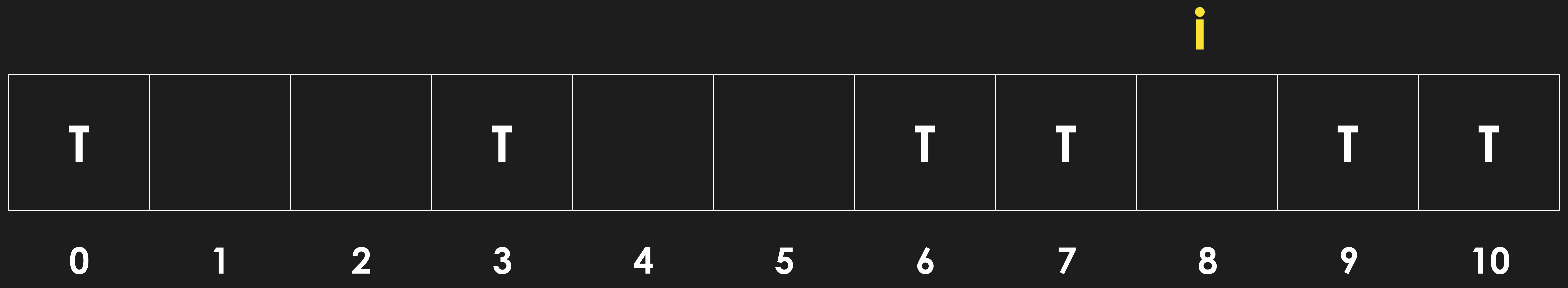
```
For i in range(target):
    if i is True:
        for n in array:
            i + n = True
```

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate



T			T			T	T		T	T
0	1	2	3	4	5	6	7	8	9	10

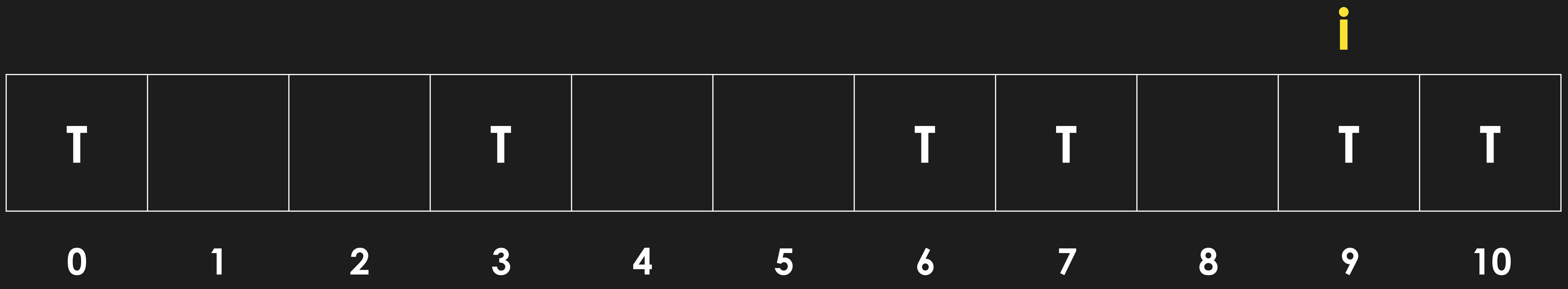
```
For i in range(target):
    if i is True:
        for n in array:
            i + n = True
```

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate



T			T			T	T		T	T
0	1	2	3	4	5	6	7	8	9	10

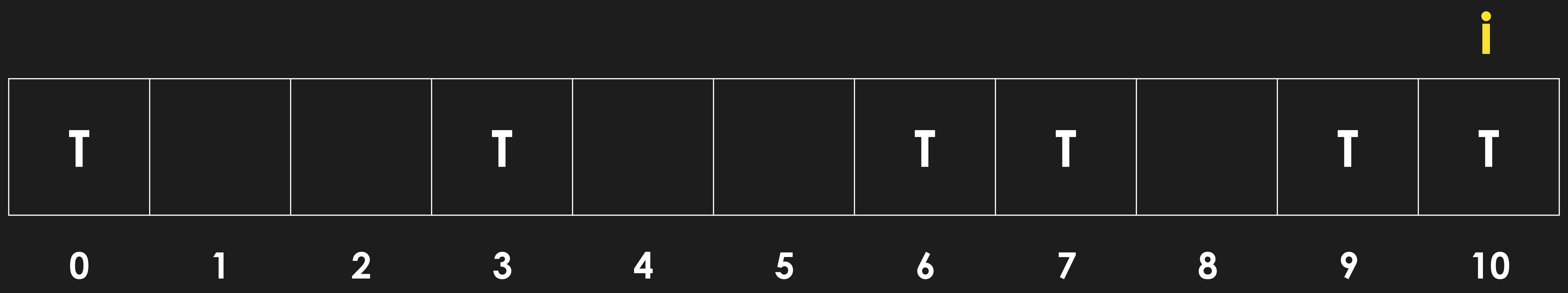
```
For i in range(target):
    if i is True:
        for n in array:
            i + n = True
```

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate



T			T			T	T		T	T
0	1	2	3	4	5	6	7	8	9	10

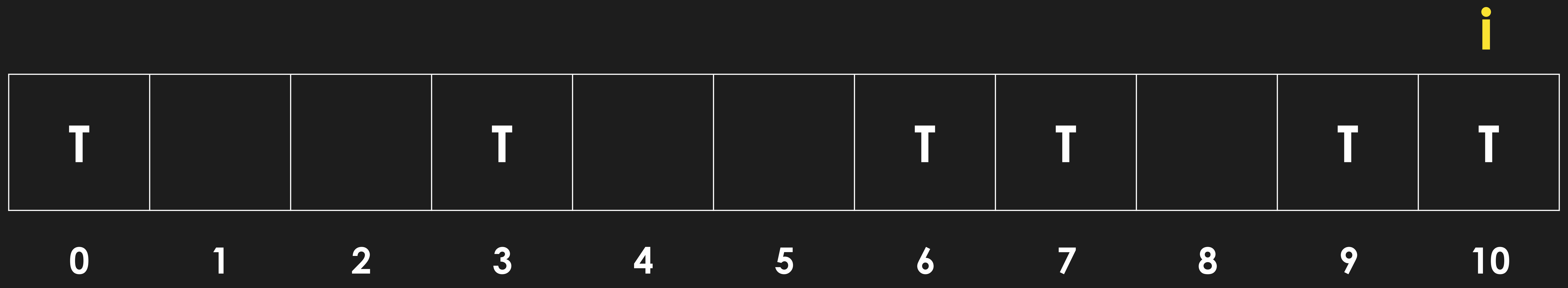
```
For i in range(target):
    if i is True:
        for n in array:
            i + n = True
```

canSum

array: [3, 6, 7]
target: 10

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

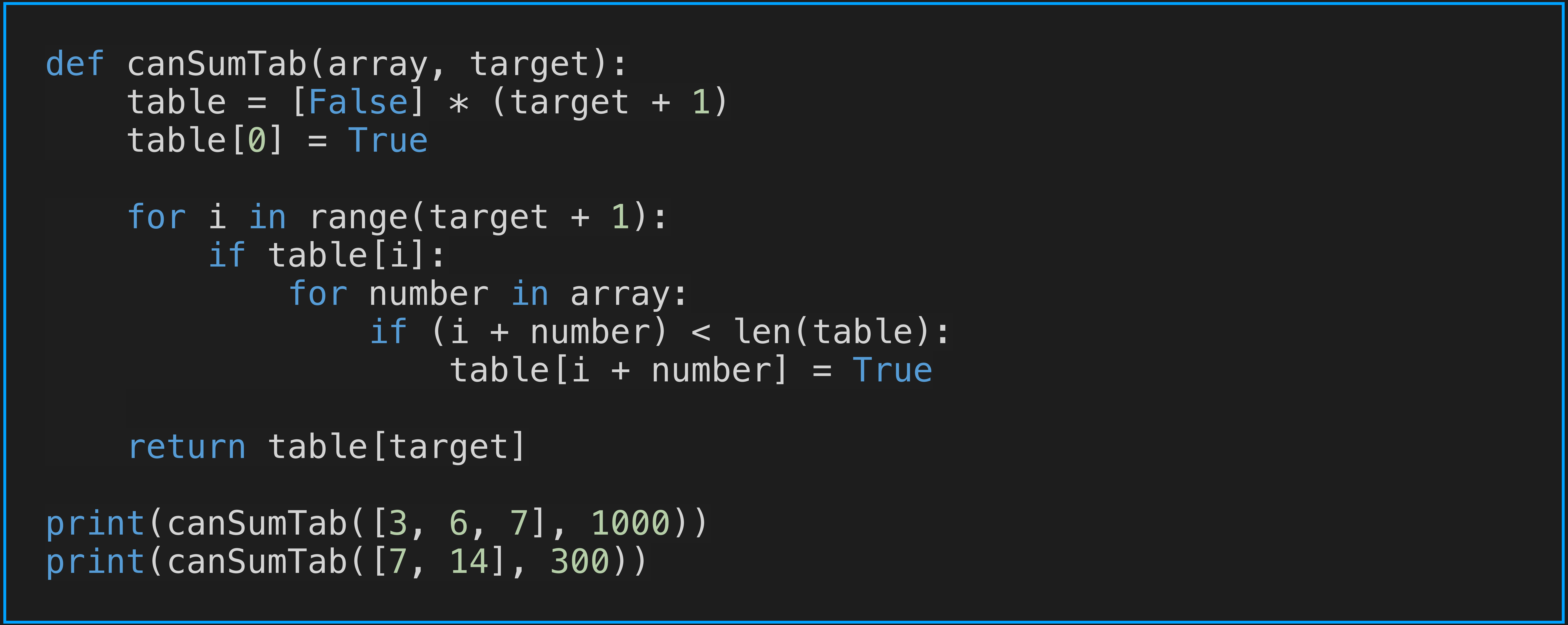


T			T			T	T		T	T
0	1	2	3	4	5	6	7	8	9	10

```
For i in range(target):
    if i is True:
        for n in array:
            i + n = True
```

time complexity: target
space complexity: target

canSumTab



```python
def canSumTab(array, target):
    table = [False] * (target + 1)
    table[0] = True

    for i in range(target + 1):
        if table[i]:
            for number in array:
                if (i + number) < len(table):
                    table[i + number] = True

    return table[target]

print(canSumTab([3, 6, 7], 1000))
print(canSumTab([7, 14], 300))
```

Problem #4: String Construct (Tabulation)

Given a target string, and an array of strings, determine whether the array of strings **can be used to construct the target string**, returning the combination with the **least number of strings**

Example:

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

stringConstruct(array, target) => ["HOL", "LER"]

Problem #4: String Construct (Tabulation)

Given a target string, and an array of strings, determine whether the array of strings **can be used to construct the target string**, returning the combination with the **least number of strings**

Initialising the table:

1. How many **variables** are being changed in each recursive call?
2. What are the **total number of "states"** that the changed variable can be in?

Problem #4: String Construct (Tabulation)

Given a target string, and an array of strings, determine whether the array of strings **can be used to construct the target string**, returning the combination with the **least number of strings**

Initialising the table:

1. How many **variables** are being changed in each recursive call? **1 (target)**
2. What are the **total number of "states"** that the changed variable can be in? **len(target)**

String Construct

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

String Construct

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

None	None	None	None	None	None	None
0	1	2	3	4	5	6

String Construct

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

[ ]	None	None	None	None	None	None
0	1	2	3	4	5	6

String Construct

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

[ ]	None	None	None	None	None	None
0	1	2	3	4	5	6

```
For i in range(len(table)):
    if table[i] is not None:
        for str in array:
            k = len(string)
            current = table[i]
            current.append(str)
            if str is prefix and len(table[i + k] > len(current):
                table[i + k] = current
```

String Construct

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

i

[ ]	**None**	**None**	**None**	**None**	**None**	**None**
0	**1**	**2**	**3**	**4**	**5**	**6**

```
For i in range(len(table)):
    if table[i] is not None:
        for str in array:
            k = len(string)
            current = table[i]
            current.append(str)
            if str is prefix and len(table[i + k] > len(current):
                table[i + k] = current
```

String Construct

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

i						
[ ]	**None**	["HO"]	["HOL"]	**None**	**None**	**None**
0	**1**	**2**	**3**	**4**	**5**	**6**

```
For i in range(len(table)):
    if table[i] is not None:
        for str in array:
            k = len(string)
            current = table[i]
            current.append(str)
            if str is prefix and len(table[i + k] > len(current):
                table[i + k] = current
```

String Construct

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

[ ]	None	["HO"]	["HOL"]	None	None	None
0	1	2	3	4	5	6

```
For i in range(len(table)):
    if table[i] is not None:
        for str in array:
            k = len(string)
            current = table[i]
            current.append(str)
            if str is prefix and len(table[i + k] > len(current):
                table[i + k] = current
```

String Construct

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

		i				
[ ]	**None**	["HO"]	["HOL"]	**None**	**None**	**None**
0	**1**	**2**	**3**	**4**	**5**	**6**

```
For i in range(len(table)):
    if table[i] is not None:
        for str in array:
            k = len(string)
            current = table[i]
            current.append(str)
            if str is prefix and len(table[i + k] > len(current):
                table[i + k] = current
```

String Construct

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

[ ]	None	["HO"]	["HOL"]	None	["HO", "LLE"]	None
0	1	2	3	4	5	6

```
For i in range(len(table)):
    if table[i] is not None:
        for str in array:
            k = len(string)
            current = table[i]
            current.append(str)
            if str is prefix and len(table[i + k] > len(current):
                table[i + k] = current
```

String Construct

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

			i			
[ ]	**None**	["HO"]	["HOL"]	**None**	["HO", "LLE"]	**None**
0	**1**	**2**	**3**	**4**	**5**	**6**

```
For i in range(len(table)):
    if table[i] is not None:
        for str in array:
            k = len(string)
            current = table[i]
            current.append(str)
            if str is prefix and len(table[i + k] > len(current):
                table[i + k] = current
```

String Construct

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

			i			
[ ]	**None**	["HO"]	["HOL"]	["HOL", "L"]	["HO", "LLE"]	["HOL", "LER"]
0	**1**	**2**	**3**	**4**	**5**	**6**

```
For i in range(len(table)):
    if table[i] is not None:
        for str in array:
            k = len(string)
            current = table[i]
            current.append(str)
            if str is prefix and len(table[i + k] > len(current):
                table[i + k] = current
```

String Construct

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

				i		
[ ]	**None**	["HO"]	["HOL"]	["HOL", "L"]	["HO", "LLE"]	["HOL", "LER"]
0	**1**	**2**	**3**	**4**	**5**	**6**

```
For i in range(len(table)):
    if table[i] is not None:
        for str in array:
            k = len(string)
            current = table[i]
            current.append(str)
            if str is prefix and len(table[i + k] > len(current):
                table[i + k] = current
```

String Construct

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

					i	
[ ]	**None**	["HO"]	["HOL"]	["HOL", "L"]	["HO", "LLE"]	["HOL", "LER"]
0	**1**	**2**	**3**	**4**	**5**	**6**

```
For i in range(len(table)):
    if table[i] is not None:
        for str in array:
            k = len(string)
            current = table[i]
            current.append(str)
            if str is prefix and len(table[i + k] > len(current):
                table[i + k] = current
```

String Construct

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

i

[ ]	**None**	["HO"]	["HOL"]	["HOL", "L"]	["HO", "LLE"]	["HOL", "LER"]
0	**1**	**2**	**3**	**4**	**5**	**6**

```
For i in range(len(table)):
    if table[i] is not None:
        for str in array:
            k = len(string)
            current = table[i]
            current.append(str)
            if str is prefix and len(table[i + k] > len(current):
                table[i + k] = current
```

stringConstructTab

```
def stringConstructTab(array, targetString):
    m = len(targetString)
    table = [None] * (m + 1)
    table[0] = []


    for i in range(m + 1):   ← m
        if table[i] != None:
            for str in array:   ← n
                j = targetString[i:].find(str)
                k = len(str)

                if j == 0 and (i + k) < m + 1:
                    newRes = table[i].copy()   ← m
                    newRes.append(str)
                    if table[i + k] == None or len(newRes) < len(table[i + k]):
                        table[i + k] = newRes

    return table[m]
```

String Construct

array = ["HO", "HOL", "L", "ER", "LER", "LLE"]
target = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

i

[ ]	**None**	["HO"]	["HOL"]	["HOL", "L"]	["HO", "LLE"]	["HOL", "LER"]
0	**1**	**2**	**3**	**4**	**5**	**6**

```
For i in range(len(table)):
    if table[i] is not None:
        for str in array:
            k = len(string)
            current = table[i]
            current.append(str)
            if str is prefix and len(table[i + k] > len(current):
                table[i + k] = current
```

len(target): m, len(array): n
time complexity: $m^2 * n$
space complexity: m

Problem #5: Longest Common Subsequence (Tabulation)

Given two strings, find the longest subsequence present in both strings

Example:

string1 = "HELLO"
string2 = "HOLLER"

lcs(string1, string2) => "HLL"

Problem #5: Longest Common Subsequence (Tabulation)

Given two strings, find the longest subsequence present in both strings

Initialising the table:

1. How many **variables** are being changed in each recursive call?
2. What are the **total number of "states"** that the changed variable can be in?

Problem #5: Longest Common Subsequence (Tabulation)

Given two strings, find the longest subsequence present in both strings

Initialising the table:

1. How many **variables** are being changed in each recursive call? **2 (string1 & string2)**
2. What are the **total number of "states"** that the changed variable can be in? **len(string1) & len(string2)**

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""						
1	"H"						
2	"O"						
3	"L"						
4	"L"						
5	"E"						
6	"R"						

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""					
2	"O"	""					
3	"L"	""					
4	"L"	""					
5	"E"	""					
6	"R"	""					

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""					
2	"O"	""					
3	"L"	""					
4	"L"	""					
5	"E"	""					
6	"R"	""					

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"				
2	"O"	""					
3	"L"	""					
4	"L"	""					
5	"E"	""					
6	"R"	""					

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"			
2	"O"	""					
3	"L"	""					
4	"L"	""					
5	"E"	""					
6	"R"	""					

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"	"H"		
2	"O"	""					
3	"L"	""					
4	"L"	""					
5	"E"	""					
6	"R"	""					

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"	"H"	"H"	"H"
2	"O"	""					
3	"L"	""					
4	"L"	""					
5	"E"	""					
6	"R"	""					

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"	"H"	"H"	"H"
2	"O"	""	"H"	"H"	"H"	"H"	
3	"L"	""					
4	"L"	""					
5	"E"	""					
6	"R"	""					

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"	"H"	"H"	"H"
2	"O"	""	"H"	"H"	"H"	"H"	"HO"
3	"L"	""					
4	"L"	""					
5	"E"	""					
6	"R"	""					

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"	"H"	"H"	"H"
2	"O"	""	"H"	"H"	"H"	"H"	"HO"
3	"L"	""	"H"	"H"			
4	"L"	""					
5	"E"	""					
6	"R"	""					

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"	"H"	"H"	"H"
2	"O"	""	"H"	"H"	"H"	"H"	"HO"
3	"L"	""	"H"	"H"	"HL"		
4	"L"	""					
5	"E"	""					
6	"R"	""					

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"	"H"	"H"	"H"
2	"O"	""	"H"	"H"	"H"	"H"	"HO"
3	"L"	""	"H"	"H"	"HL"	"HL"	"HL"
4	"L"	""					
5	"E"	""					
6	"R"	""					

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"	"H"	"H"	"H"
2	"O"	""	"H"	"H"	"H"	"H"	"HO"
3	"L"	""	"H"	"H"	"HL"	"HL"	"HL"
4	"L"	""	"H"	"H"			
5	"E"	""					
6	"R"	""					

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"	"H"	"H"	"H"
2	"O"	""	"H"	"H"	"H"	"H"	"HO"
3	"L"	""	"H"	"H"	"HL"	"HL"	"HL"
4	"L"	""	"H"	"H"	"HL"		
5	"E"	""					
6	"R"	""					

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"	"H"	"H"	"H"
2	"O"	""	"H"	"H"	"H"	"H"	"HO"
3	"L"	""	"H"	"H"	"HL"	"HL"	"HL"
4	"L"	""	"H"	"H"	"HL"	"HLL"	
5	"E"	""					
6	"R"	""					

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"	"H"	"H"	"H"
2	"O"	""	"H"	"H"	"H"	"H"	"HO"
3	"L"	""	"H"	"H"	"HL"	"HL"	"HL"
4	"L"	""	"H"	"H"	"HL"	"HLL"	"HLL"
5	"E"	""					
6	"R"	""					

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"	"H"	"H"	"H"
2	"O"	""	"H"	"H"	"H"	"H"	"HO"
3	"L"	""	"H"	"H"	"HL"	"HL"	"HL"
4	"L"	""	"H"	"H"	"HL"	"HLL"	"HLL"
5	"E"	""	"H"	"H"	"HL"	"HLL"	"HLL"
6	"R"	""					

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"	"H"	"H"	"H"
2	"O"	""	"H"	"H"	"H"	"H"	"HO"
3	"L"	""	"H"	"H"	"HL"	"HL"	"HL"
4	"L"	""	"H"	"H"	"HL"	"HLL"	"HLL"
5	"E"	""	"H"	"H"	"HL"	"HLL"	"HLL"
6	"R"	""	"H"	"H"	"HL"	"HLL"	"HLL"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"	"H"	"H"	"H"
2	"O"	""	"H"	"H"	"H"	"H"	"HO"
3	"L"	""	"H"	"H"	"HL"	"HL"	"HL"
4	"L"	""	"H"	"H"	"HL"	"HLL"	"HLL"
5	"E"	""	"H"	"H"	"HL"	"HLL"	"HLL"
6	"R"	""	"H"	"H"	"HL"	"HLL"	🟩 "HLL"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

Earlier, we learnt that the optimal substructure of the LCS problem was as such:

- **case 1:** characters match
- **case 2:** characters don't match

This means that for some cell (i, j), the LCS is equals to:

- **From case 1:** cell (i - 1, j - 1) + **matched char (if matched)**
- **From case 2:** cell (i - 1, j) OR cell (i, j - 1)

Whichever has the greatest length

Longest Common Subsequence

string1 = "HELLO"
string2 = "HOLLER"

string1

string2

		0	1	2	3	4	5
		""	"H"	"E"	"L"	"L"	"O"
0	""	""	""	""	""	""	""
1	"H"	""	"H"	"H"	"H"	"H"	"H"
2	"O"	""	"H"	"H"	"H"	"H"	"HO"
3	"L"	""	"H"	"H"	"HL"	"HL"	"HL"
4	"L"	""	"H"	"H"	"HL"	"HLL"	"HLL"
5	"E"	""	"H"	"H"	"HL"	"HLL"	"HLL"
6	"R"	""	"H"	"H"	"HL"	"HLL"	"HLL"

Pseudo Steps

1. Initialise table
2. Fill up base cases
3. Iterate

len(string1): **m**, len(string2): **n**

time complexity: $m * n * max(m, n)$
space complexity: $m * n$

LCS Tab Implementation

```python
def maxLengthString(prev, top, left):
    res = top if len(top) > len(left) else left
    res = res if len(res) > len(prev) else prev
    return res

def lcsTab(string1, string2):
    m = len(string1)
    n = len(string2)

    table = [["" for j in range(m + 1)] for i in range(n + 1)]

    for i in range(n+ 1):
        table[i][0] = ""
    for j in range(m + 1):
        table[0][j] = ""

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            prev = table[i - 1][j - 1]

            if string2[i - 1] == string1[j - 1]:
                prev += string1[j - 1]

            top = table[i - 1][j]
            left = table[i][j - 1]

            table[i][j] = maxLengthString(prev, top, left)

    for row in table:
        print(row)
    return table[n][m]
```

Lab Session 2

Lab Session 2

- In this lab session, you will be implementing **tabulation.py**
- Your task is to implement **the following functions:**

1. **fib:**
    - Takes in an int n and returns the nth number in the fibonacci sequence
    - This function should run in **O(N) time**

2. **bestSum:**
    - Takes in an array of integers and a target sum
    - Should return an array of any combination of integers included in the array, with repeats allowed, which sum to the target
    - This array should be the of the smallest size possible, or empty if there is no solution
    - Should run in **$O(NM^2)$ time**, where N is the length of the array and M is the target sum

Lab Session 2

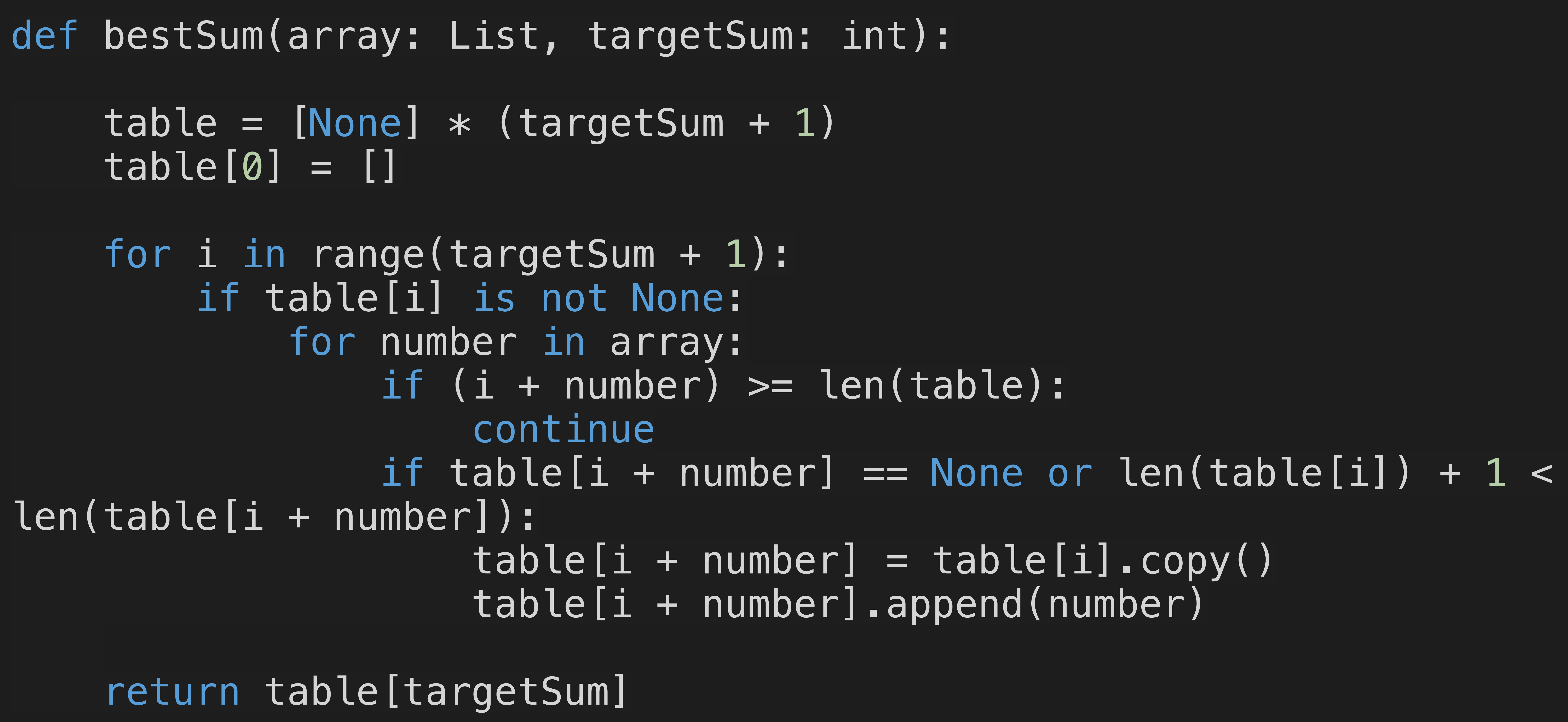
3. **longestCommonSubsequence:**
    - Takes in two strings string1 & string2
    - Returns the longest common subsequence between string1 & string2 as a string, or an empty string if there is none
    - Should run in **O(NM) time**

- All three problems should be done using the tabulation method
- To test the problems, run **`python utils/tab_test.py`**

Solution: fib

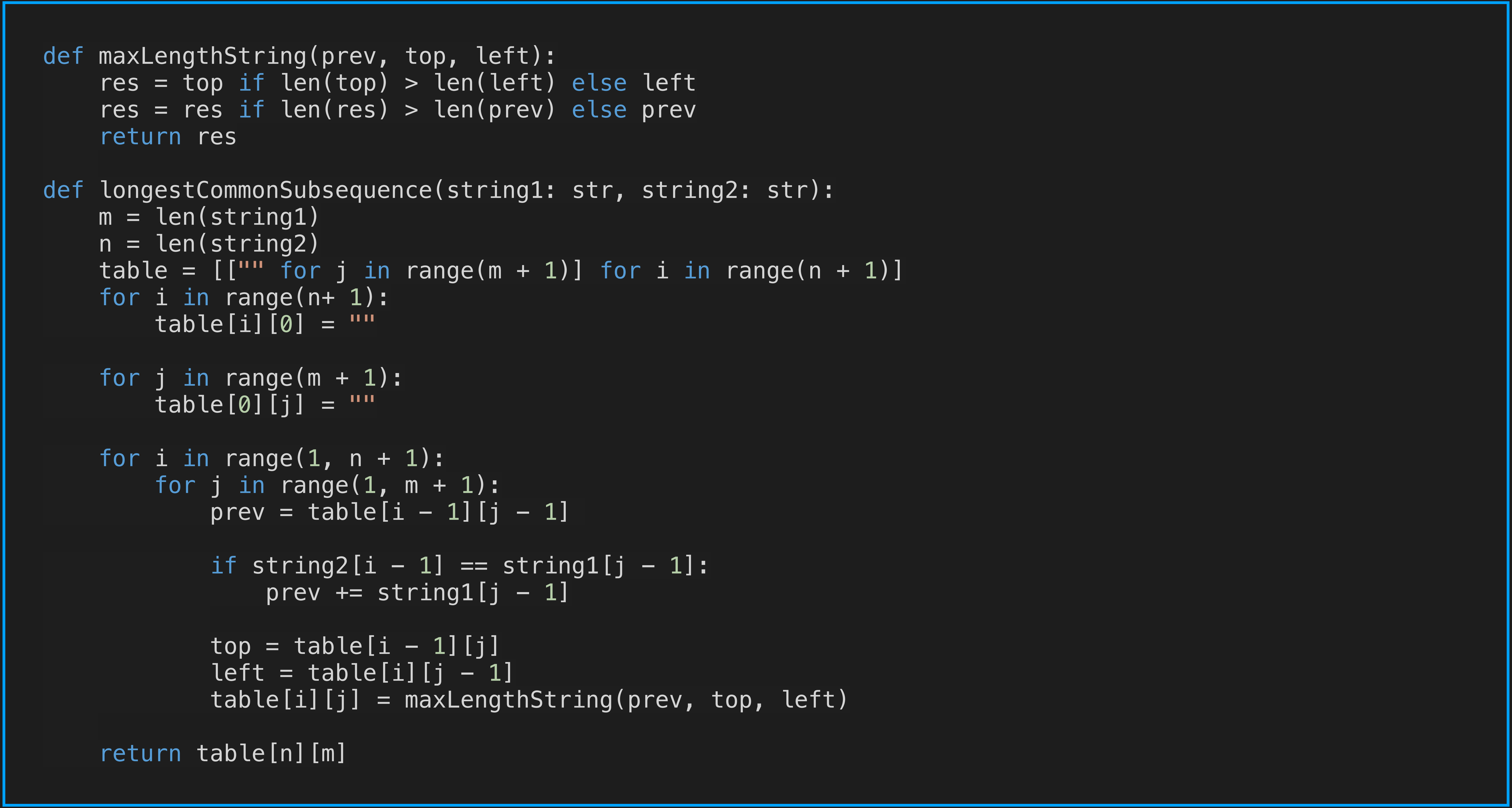
```python
def fib(n: int):

    table = [None] * (n + 1)

    table[0] = 1
    table[1] = 1

    for i in range(2, n + 1):
        table[i] = table[i - 1] + table[i - 2]

    return table[n]
```

Solution: bestSum



```
def bestSum(array: List, targetSum: int):

    table = [None] * (targetSum + 1)
    table[0] = []

    for i in range(targetSum + 1):
        if table[i] is not None:
            for number in array:
                if (i + number) >= len(table):
                    continue
                if table[i + number] == None or len(table[i]) + 1 <
len(table[i + number]):
                    table[i + number] = table[i].copy()
                    table[i + number].append(number)

    return table[targetSum]
```

Solution: longestCommonSubsequence



```python
def maxLengthString(prev, top, left):
    res = top if len(top) > len(left) else left
    res = res if len(res) > len(prev) else prev
    return res

def longestCommonSubsequence(string1: str, string2: str):
    m = len(string1)
    n = len(string2)
    table = [["" for j in range(m + 1)] for i in range(n + 1)]
    for i in range(n+ 1):
        table[i][0] = ""

    for j in range(m + 1):
        table[0][j] = ""

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            prev = table[i - 1][j - 1]

            if string2[i - 1] == string1[j - 1]:
                prev += string1[j - 1]

            top = table[i - 1][j]
            left = table[i][j - 1]
            table[i][j] = maxLengthString(prev, top, left)

    return table[n][m]
```

Conclusion

Understanding a DP Problem:

Memoization:

1. **Identify the optimal substructure:**
   - How can we break the larger problem into subproblems?
   - How can we use the solutions to sub problems to solve the larger problem?

2. **Identify the overlapping subproblems:**
   - Which variables are being changed through each recursive call?

Tabulation:

1. **Identify the table structure of the problem:**
   - How many variables are being changed: number of dimensions for the table
   - How many states can each variable take?

2. **Identify the iterative nature:**
   - How can we use subproblems to contribute a value to the larger problem?

Memoization vs Tabulation

Memoization	**Tabulation**
Better if only some of the subproblems are required to be solved	Can solve all sub problems through an iterative method
Difficult to analyse time & space complexities	Easy to analyse time & space complexities
Recursive stack calling is slower	Simple iteration is faster

Memoization vs Tabulation

- It is worth to note however, that for **tabulation**, to understand the iterative nature, you have to understand the optimal substructure property of the problem, which is usually easier to visualise in a recursive nature, before applying to the tabular structure.

- As such, the **memoization** approach may be more intuitive in nature and tabular solutions may be more confusing and difficult to understand because of the failure to demonstrate optimal substructure

Applications of DP

1. Travelling Salesman Problem
2. Levenshtein Distance (String Matching)
3. Flight & Robotics Control