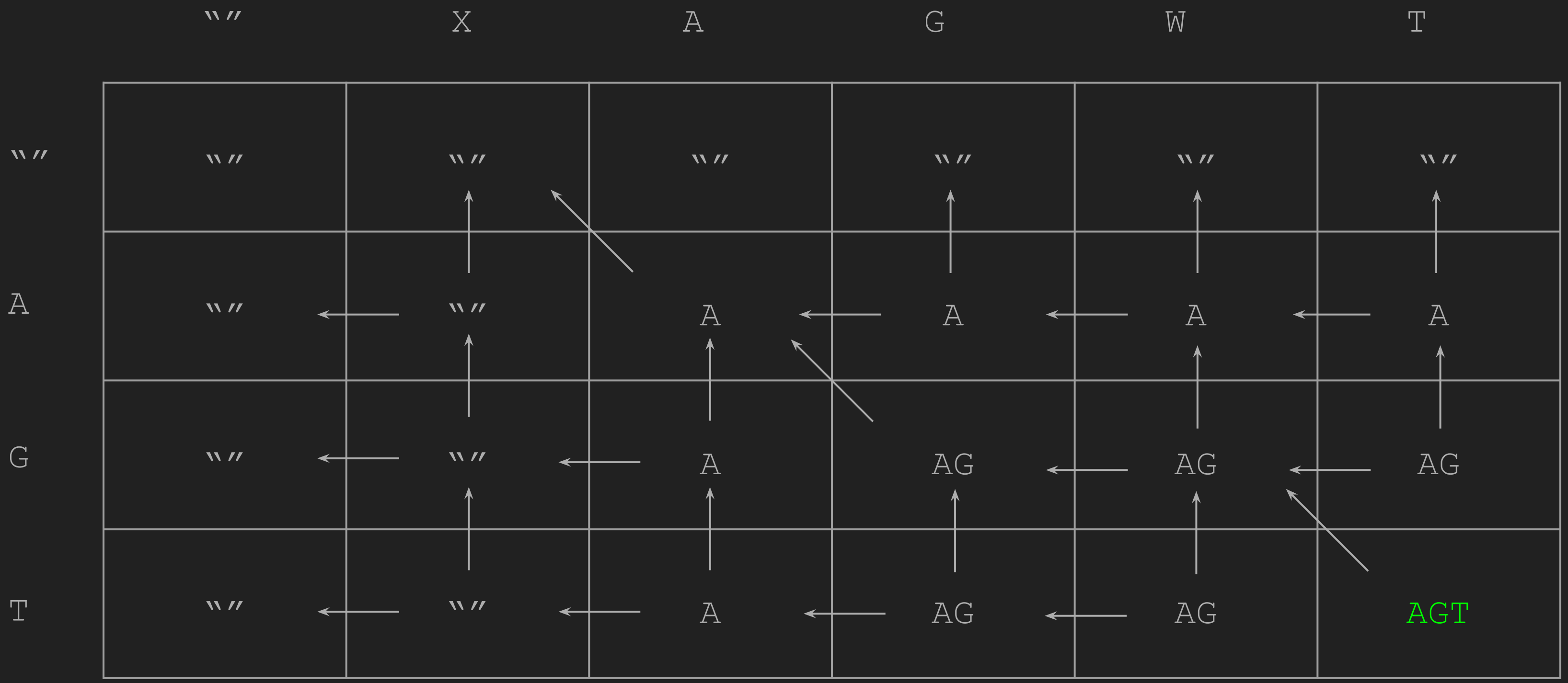
Dynamic Programming

LCS: Alternative implementation with “table-filling”

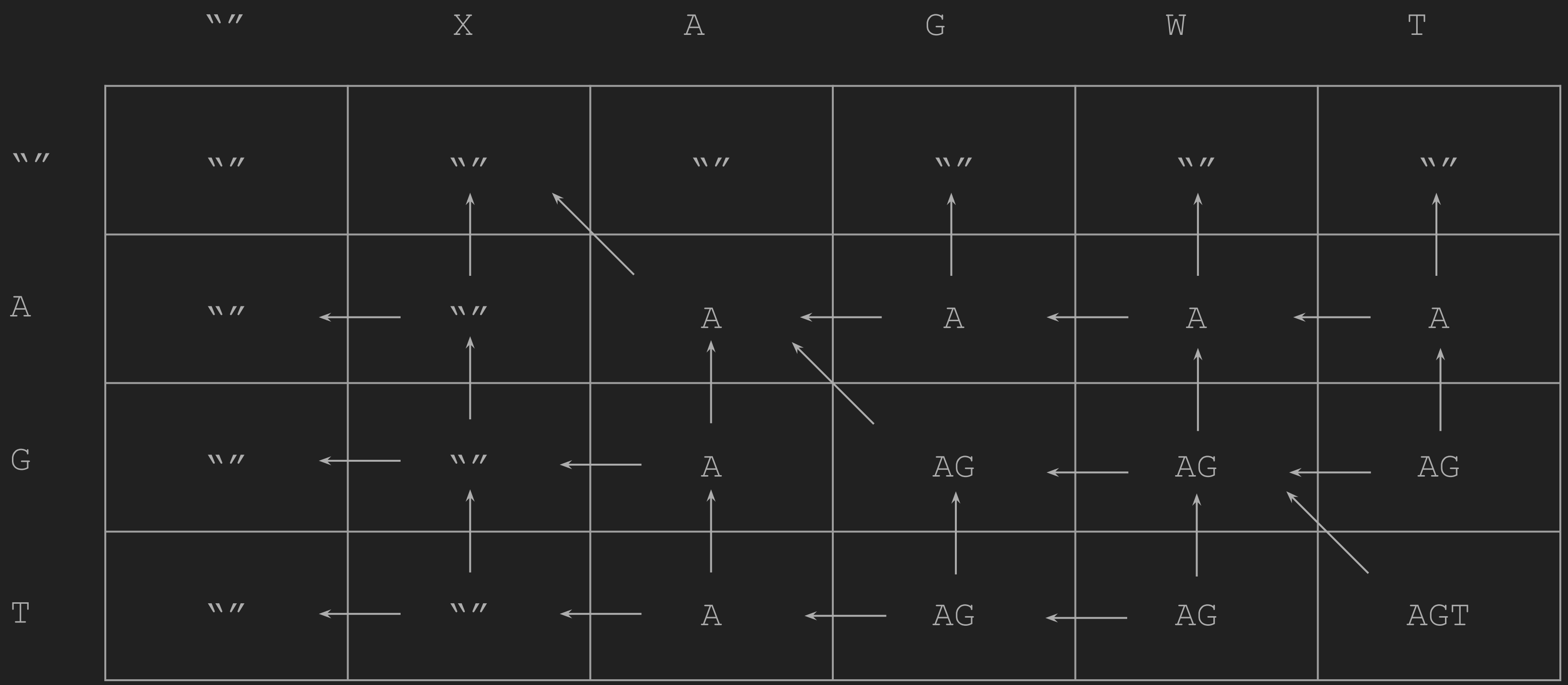


	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A	AG	AG	AG
T	“”	“”	A	AG	AG	AGT

DP Example: Longest common subsequence

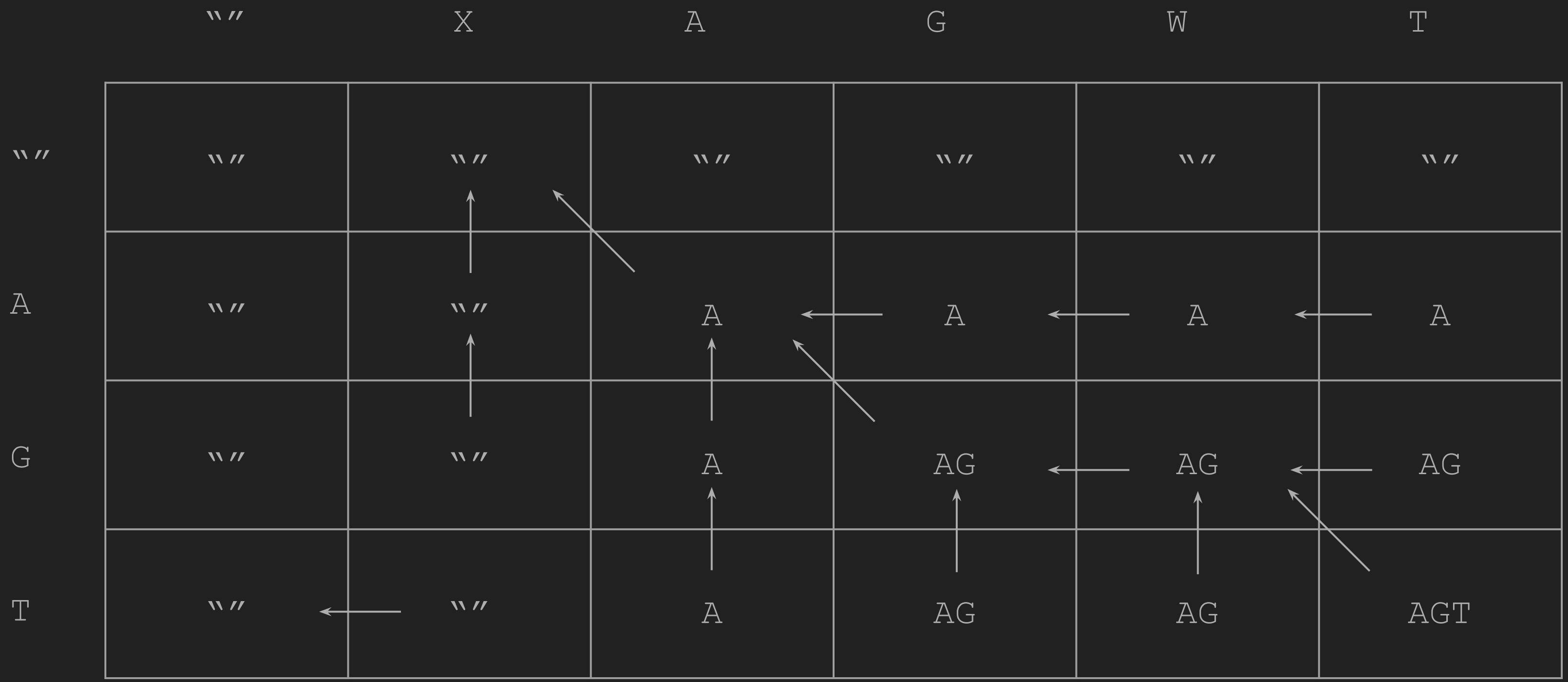
- Iterative “table-filling” runtime complexity
    - Filling in an n * m grid, so *O(nm)*
    - Space is worse, because we’re storing the whole string
        - Can improve by only storing the path to the previous call, and reconstruct answer later

LCS: Space saving with “table-filling”



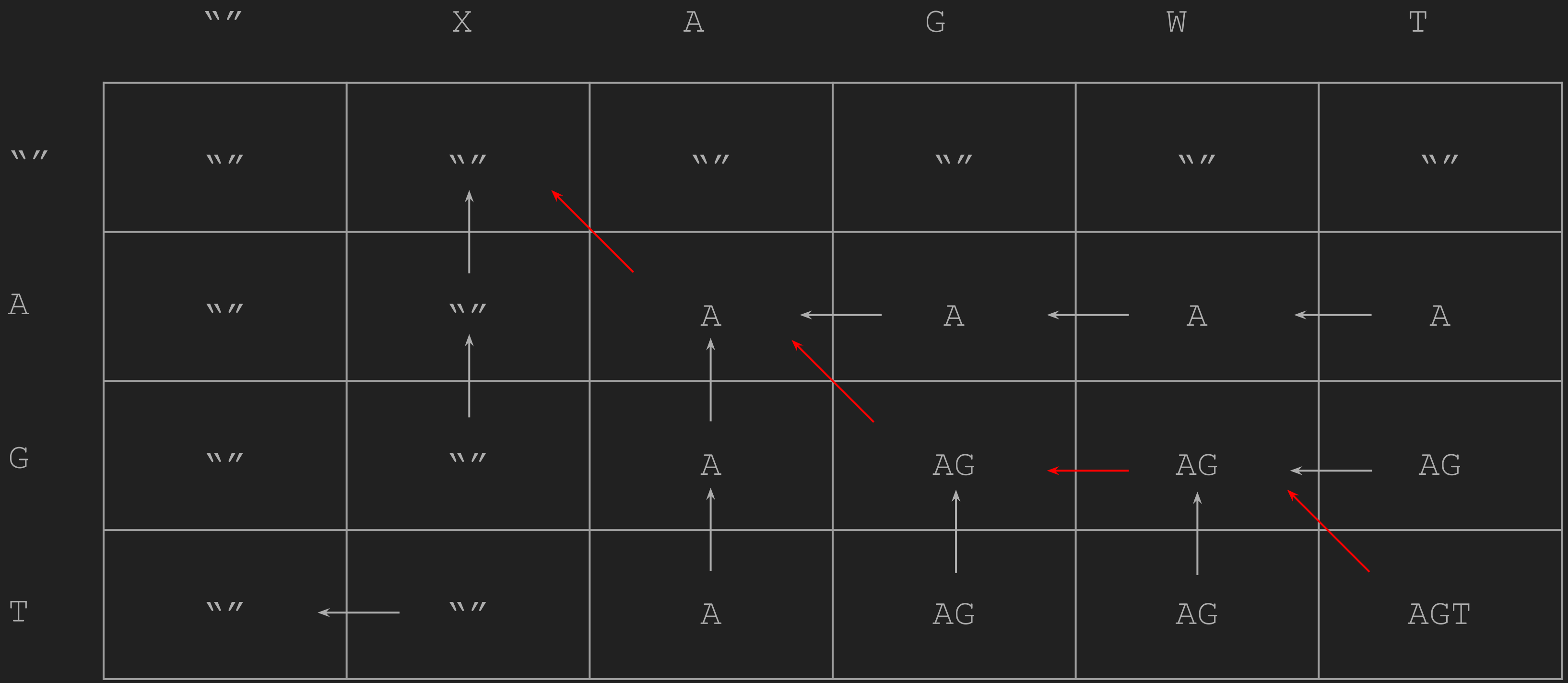
	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A	AG	AG	AG
T	“”	“”	A	AG	AG	AGT

LCS: Space saving with “table-filling”



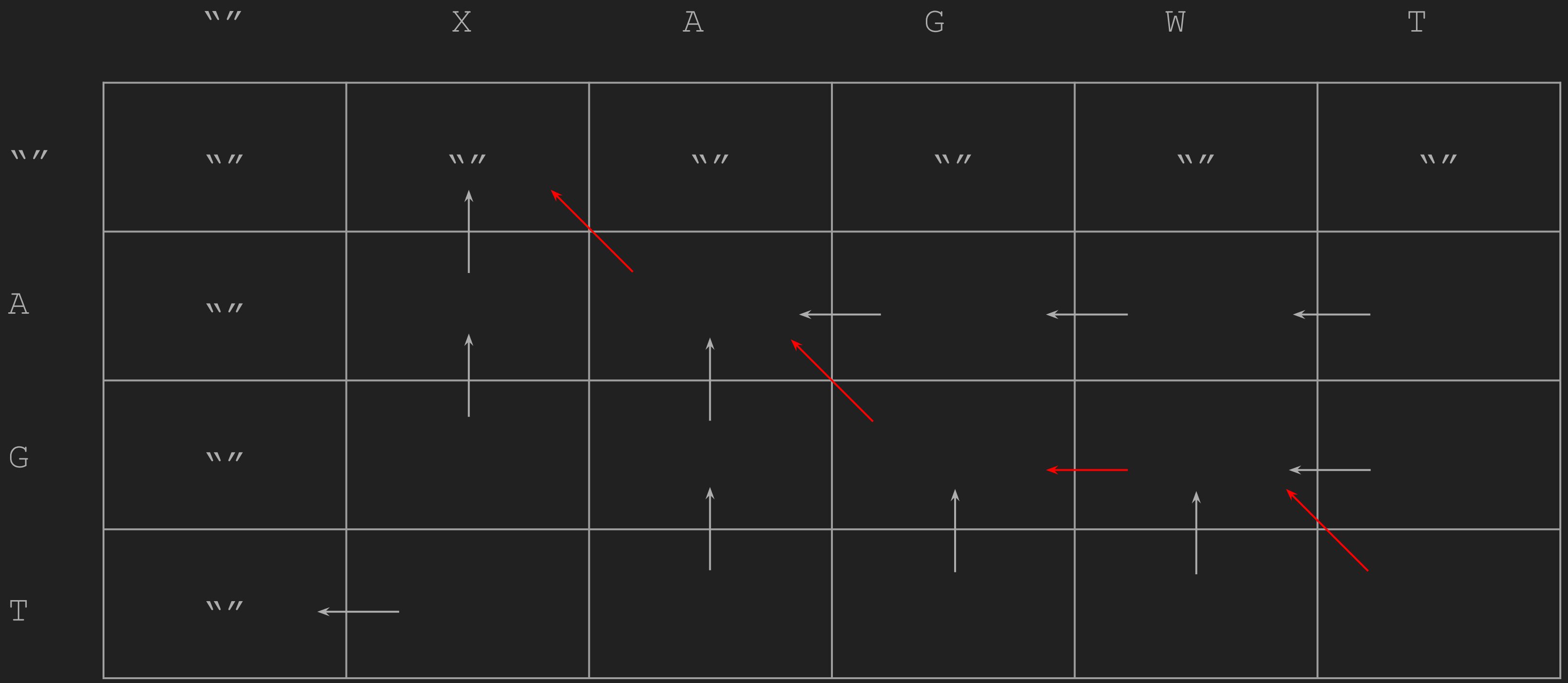
	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A	AG	AG	AG
T	“”	“”	A	AG	AG	AGT

LCS: Space saving with “table-filling”



	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”	“”	A	A	A	A
G	“”	“”	A	AG	AG	AG
T	“”	“”	A	AG	AG	AGT

LCS: Space saving with “table-filling”



	“”	X	A	G	W	T
“”	“”	“”	“”	“”	“”	“”
A	“”					
G	“”					
T	“”					

DP Example: Longest common subsequence

- Iterative “table-filling” runtime complexity
  - Filling in an n * m grid, so *O(nm)*
  - Space is worse, because we’re storing the whole string
    - Can improve by only storing the path to the previous call, and reconstruct answer later
- Recursive memoization runtime complexity
  - Essentially memoizing values for the cells visited
  - *O(nm)* still a reasonable upper bound
  - Space can be improved in a similar way
- Practical applications
  - diff
  - version control systems
  - bioinformatics
  - computational linguistics

LCS application: diff

Sequence 1: A B D F H Y Z

Sequence 2: A B C F H W X Y Z

LCS application: diff

Sequence 1: A B D F H Y Z

Sequence 2: A B C F H W X Y Z

LCS: A B F H Y Z

diff: D C W X
- \+ + +

DP Example: Maximum subarray problem

- Given: Array *a* containing integers $[x_1, \ldots, x_n]$
- Find: integers i, j such that $1 \leq i \leq j \leq n$ and

$$\sum_{t=i}^{j} x_t \quad \text{is maximal}$$

- Brute force: try all (i, j) pairs (where $i \leq j$)
  - Runtime complexity: $O(n^3)$

DP Example: Maximum subarray problem

- What is the optimal substructure?
- First idea: optimal solution to $a[x_1,\ldots,x_n]$ is the optimal solution to $a[x_1,\ldots,x_{n-1}]$ plus the decision to add in or leave out x_n
    - Doesn’t quite work, consider [100, -10, 50]
    - Optimal solution to [100, -10] is 100
    - If we add in the 50 we have 150
    - But wait… 100 and 50 aren’t consecutive! The real optimal is 140
- Insight: we may need to include negative numbers in our running sum

DP Example: Maximum subarray problem

- Insight: the maximum subarray will have to end in some element x_i
- So lets reframe the problem a little: what's the maximum subarray for $a[x_i,...,x_n]$ **that includes** x_n
    - the maximum sum in $a[x_1,...,x_{n-1}]$ **that includes x_{n-1}** plus x_n, OR
    - simply x_n
- Intuition: build up sum as you go, but if sum would ever be lower than the next value alone you can "cut your losses"
    - Insures that adding in x_n is even a valid option
    - Still breaking down things into subproblems
- Can the solution to this problem be used to find the solution to our original problem?
    - Yes! The maximum sum will end in some x_i and this solution will find that sum
    - ...but this means we won't know which x_i until after we calculate all the sums

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

0	1	2	3	4	5	6	7

`mem[n] = max(mem[n-1] + a[n], a[n])`

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

0	1	2	3	4	5	6	7
-2							

`mem[n] = max(mem[n-1] + a[n], a[n])`

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

0	1	2	3	4	5	6	7
-2							

```
mem[n] = max(mem[n-1] + a[n], a[n])
```

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

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

```
mem[n] = max(mem[n-1] + a[n], a[n])
```

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

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

```
mem[n] = max(mem[n-1] + a[n], a[n])
```

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

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

`mem[n] = max(mem[n-1] + a[n], a[n])`

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

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

```
mem[n] = max(mem[n-1] + a[n], a[n])
```

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

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

```
mem[n] = max(mem[n-1] + a[n], a[n])
```

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

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

`mem[n] = max(mem[n-1] + a[n], a[n])`

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

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

```
mem[n] = max(mem[n-1] + a[n], a[n])
```

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

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

```
mem[n] = max(mem[n-1] + a[n], a[n])
```

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

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

`mem[n] = max(mem[n-1] + a[n], a[n])`

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

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

`mem[n] = max(mem[n-1] + a[n], a[n])`

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

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

```
mem[n] = max(mem[n-1] + a[n], a[n])
```

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

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

`mem[n] = max(mem[n-1] + a[n], a[n])`

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

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

```
mem[n] = max(mem[n-1] + a[n], a[n])
```

DP Example: Maximum subarray problem

[-2, -5, 6, -2, -3, 1, 5, -6]

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

`mem[n] = max(mem[n-1] + a[n], a[n])`

DP Example: Maximum subarray problem

```
def maximum_subarray(a):
    mem = [0 for x in range(len(a))]
    mem[0] = a[0]
    for i in range(1, len(a)):
        tmp = mem[i-1] + a[i]
        mem[i] = tmp if tmp > a[i] else a[i]
    max_sum = mem[0]
    for i in range(1, len(a)):
        max_sum = mem[i] if mem[i] > max_sum else max_sum
    return max_sum
```

DP Example: Maximum subarray problem

- Runtime analysis:
    - Two passes through the list, so *O(n)*

DP Example: Knapsack Problem

- Imagine you have a knapsack that can only hold up to a certain amount of weight. You want to fill it with items that cumulatively are higher value than any other items you could fill it with, while still making sure to stay under the weight limit.
- Given: a set of n items, each with value v_i and weight w_i (both integers), as well as an integer W
- Goal: choosing x_i as either 0 or 1 for each i,

$$\text{maximize} \quad \sum_{i=1}^{n} v_i x_i \qquad \text{subject to to the constraint} \qquad \sum_{i=1}^{n} w_i x_i \leq W$$

DP Example: Knapsack Problem

- Practical Applications
    - Resource allocation
        - Computer systems
        - Financial investments
        - Test Scoring
        - Cutting raw materials
        - Daily Fantasy Sports
        - etc. etc.
    - Cryptography (Merkle-Hellman)

DP Example: Knapsack Problem

- Initial insight: for each item, we either choose to put it in the knapsack or not
    - If you choose to put item *i* in the knapsack, now you have a knapsack that can hold $W - w_i$ weight and *n-1* items to fill it with
        - This is a subproblem! Fewer items, and smaller max weight
    - If you choose NOT to put item *i* in the knapsack, you now have *n-1* items to fill the knapsack (which can still hold *W* weight).
        - This is also a subproblem! Only fewer items this time.
- It seems like there's an optimal substructure: we can break the problem down into smaller subproblems
- Two different dimensions the problem can be broken down: number of items, and max weight
- Additional insight: if an item's weight is greater than *W*, we can't choose it

DP Example: Knapsack Problem

- Define `m[i, w]` to be the highest value you can obtain with the first *i* items without going over weight *w*
- $m[0, w] = 0$
- $m[i, w] = m[i - 1, w] \quad \text{if} \quad w_i > w$
- $m[i, w] = \max(m[i - 1, w], m[i - 1, w - w_i] + v_i) \quad \text{if} \quad w_i \leq w$
- Solution to problem: `m[n, W]`
- How should we fill the table?
  - Always rely on subproblems with one fewer item, so need to complete row for *i-1* before moving on to row for *i*
  - Could examine row *i-1* for any *w*, so need to calculate for all *w* before moving on

DP Example: Knapsack Problem

Items (*v*, *w*): {(3, 2), (4, 3), (5, 4), (6, 5)}

					w		
		0	1	2	3	4	5
	0						
	1						
i	2						
	3						
	4						

DP Example: Knapsack Problem

Items (*v*, *w*): {(3, 2), (4, 3), (5, 4), (6, 5)}

		w					
		0	1	2	3	4	5
	0	0	0	0	0	0	0
	1						
i	2						
	3						
	4						

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}

				w			
		0	1	2	3	4	5
i	0	0	0	0	0	0	0
	1	0					
	2	0					
	3	0					
	4	0					

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}

w

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}

w

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

DP Example: Knapsack Problem

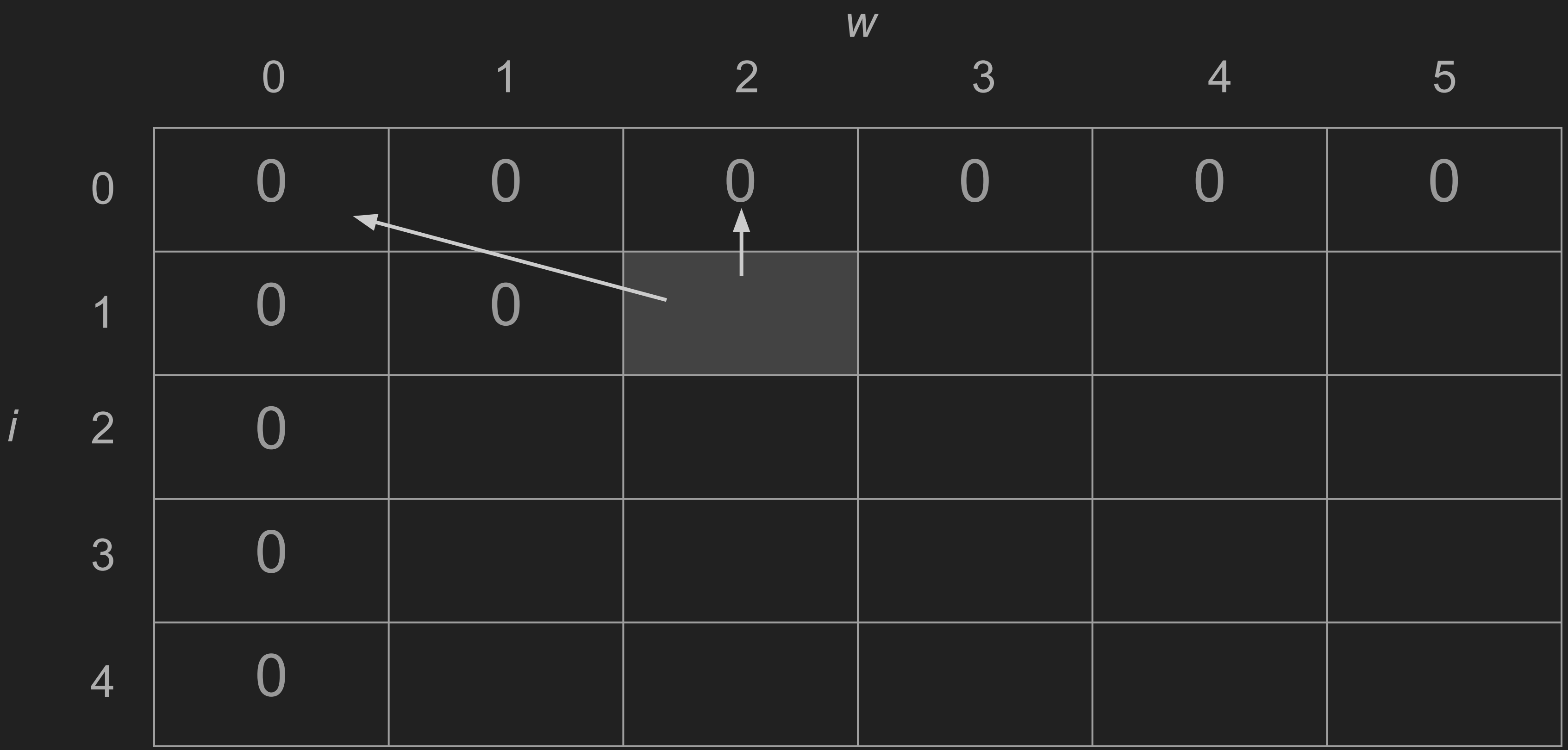
Items (*v*, *w*): {(3, 2), (4, 3), (5, 4), (6, 5)}

w

		0	1	2	3	4	5
	0	0	0	0	0	0	0
	1	0	0				
i	2	0					
	3	0					
	4	0					

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}



		w					
		0	1	2	3	4	5
i	0	0	0	0	0	0	0
	1	0	0				
	2	0					
	3	0					
	4	0					

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}

		w					
		0	1	2	3	4	5
	0	0	0	0	0	0	0
	1	0	0	3			
i	2	0					
	3	0					
	4	0					

DP Example: Knapsack Problem

Items (*v*, *w*): {(3, 2), (4, 3), (5, 4), (6, 5)}

		w 0	1	2	3	4	5
	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
i	2	0					
	3	0					
	4	0					

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}

w

		0	1	2	3	4	5
	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
i	2	0					
	3	0					
	4	0					

DP Example: Knapsack Problem

Items (v, w): $\{(3, 2), (4, 3), (5, 4), (6, 5)\}$

w

i

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

DP Example: Knapsack Problem

Items (*v*, *w*): {(3, 2), (4, 3), (5, 4), (6, 5)}

				w			
		0	1	2	3	4	5
	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
i	2	0	0				
	3	0					
	4	0					

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}

w

i

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}

		w					
		0	1	2	3	4	5
	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
i	2	0	0	3			
	3	0					
	4	0					

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}

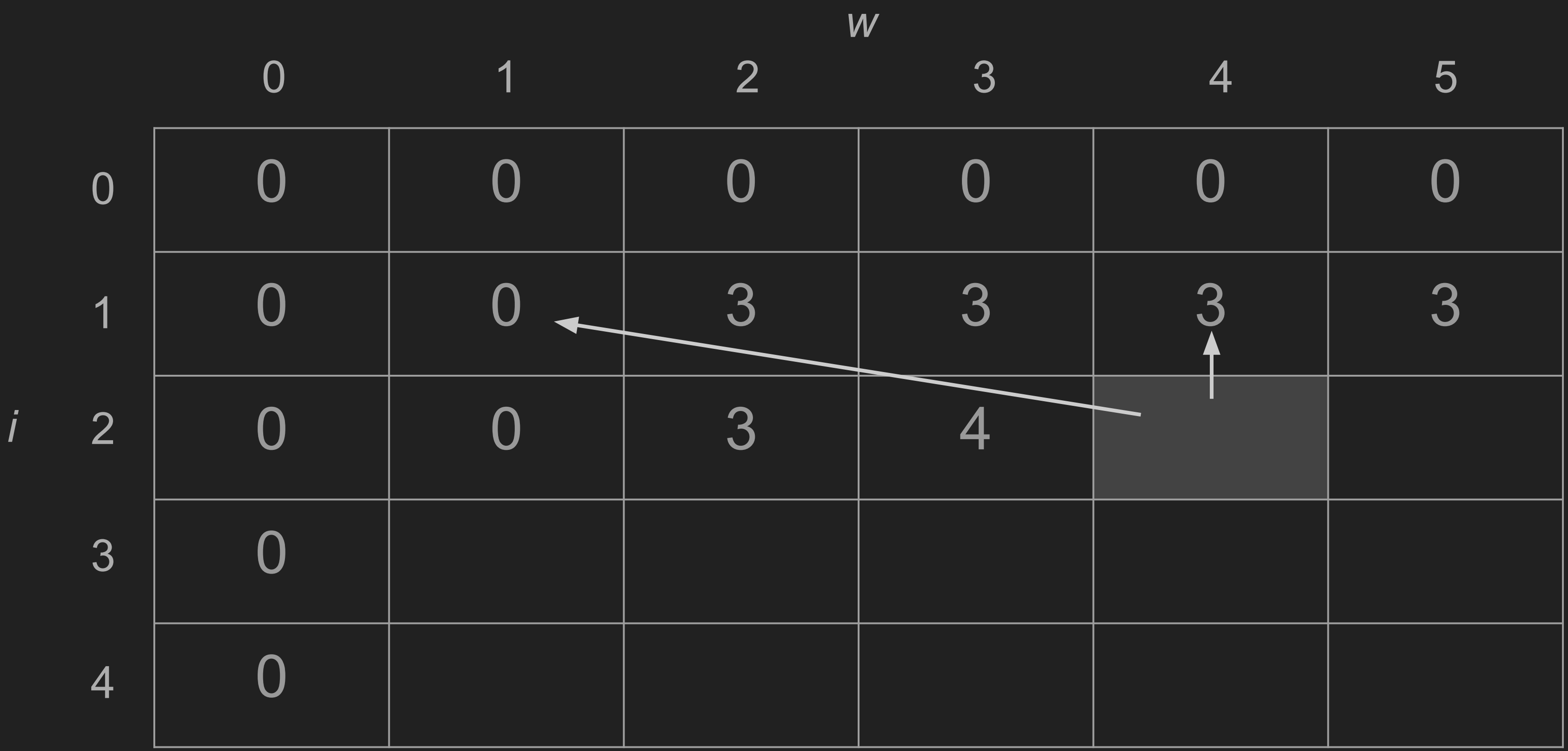
w

i

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}



w

		0	1	2	3	4	5
	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
i	2	0	0	3	4		
	3	0					
	4	0					

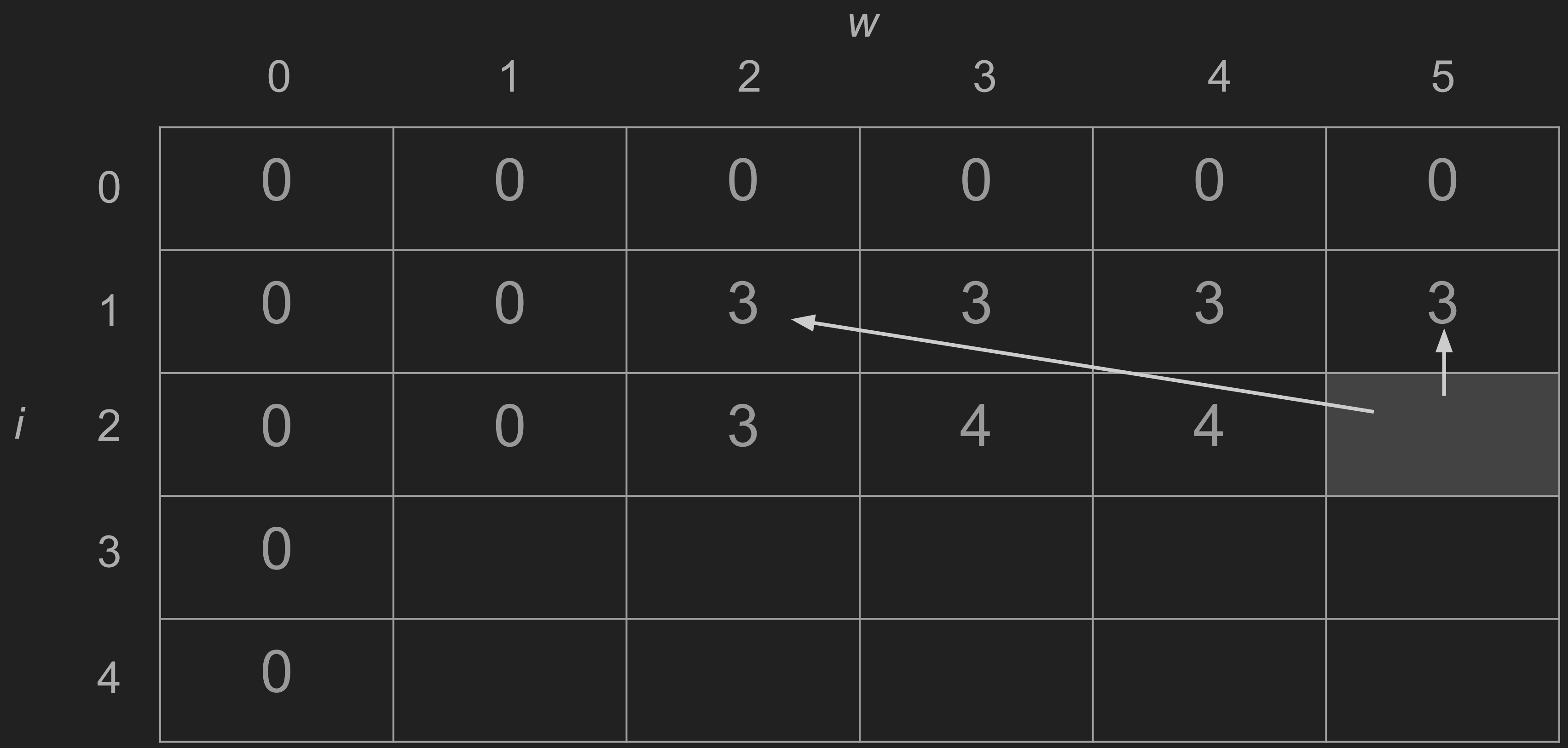
DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}

		w					
		0	1	2	3	4	5
i	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
	2	0	0	3	4	4	
	3	0					
	4	0					

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}



w

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}

		w					
		0	1	2	3	4	5
	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
i	2	0	0	3	4	4	7
	3	0					
	4	0					

DP Example: Knapsack Problem

Items (*v*, *w*): {(3, 2), (4, 3), (5, 4), (6, 5)}

					w		
		0	1	2	3	4	5
	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
i	2	0	0	3	4	4	7
	3	0					
	4	0					

DP Example: Knapsack Problem

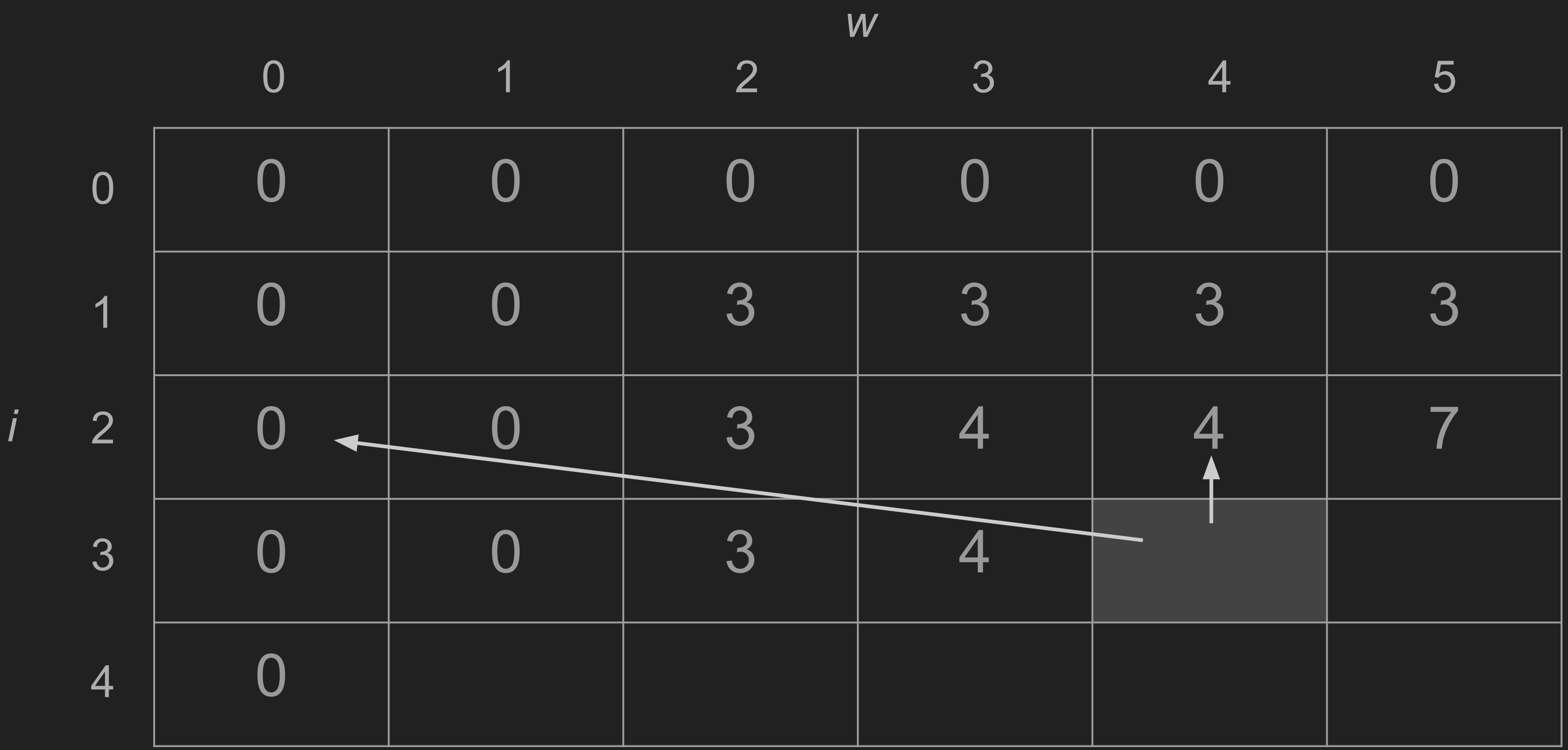
Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}

w

		0	1	2	3	4	5
	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
i	2	0	0	3	4	4	7
	3	0	0	3	4		
	4	0					

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}



	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

DP Example: Knapsack Problem

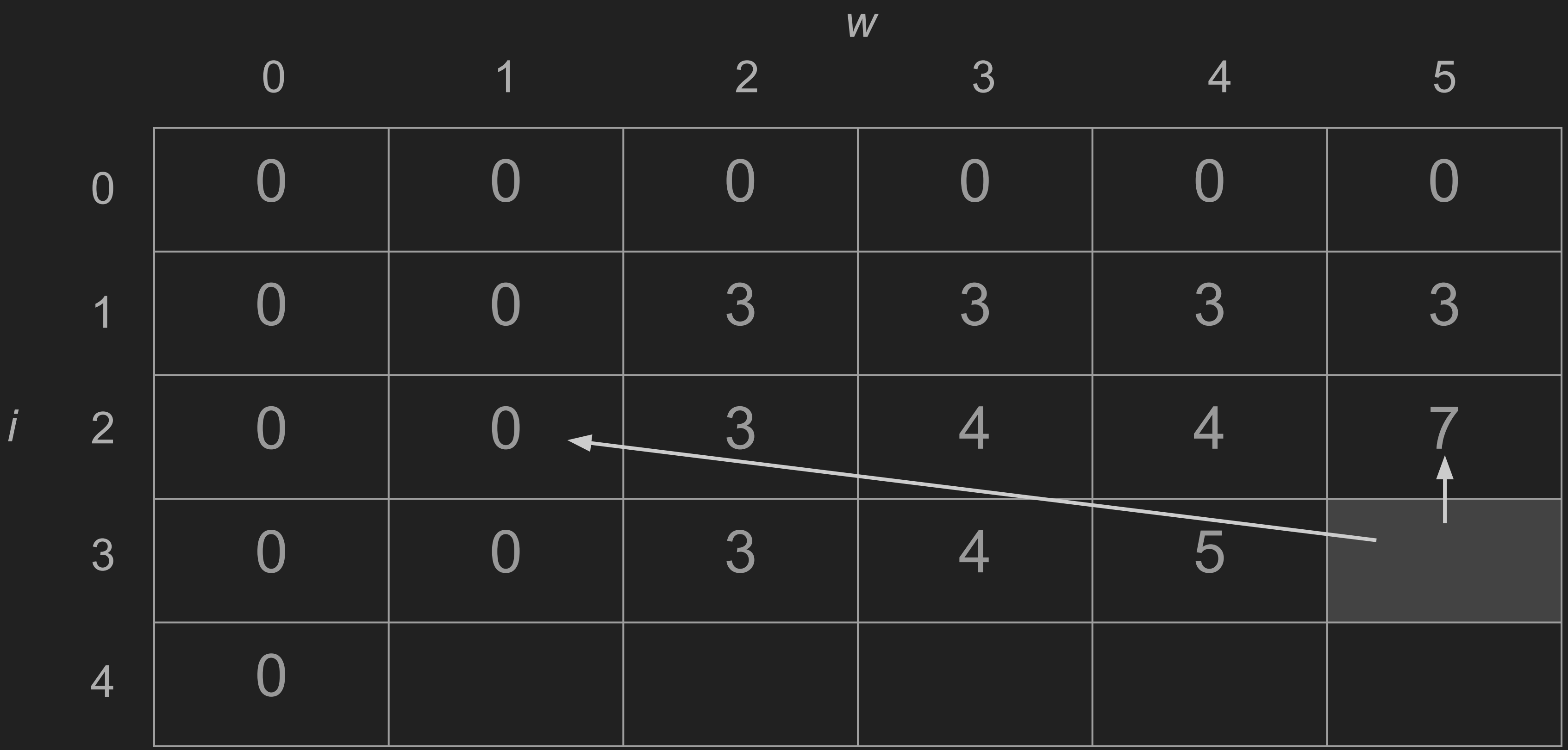
Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}

w

		0	1	2	3	4	5
	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
i	2	0	0	3	4	4	7
	3	0	0	3	4	5	
	4	0					

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}



w

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

DP Example: Knapsack Problem

Items (*v*, *w*): {(3, 2), (4, 3), (5, 4), (6, 5)}

		w					
		0	1	2	3	4	5
	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
i	2	0	0	3	4	4	7
	3	0	0	3	4	5	7
	4	0					

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}

i \ w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

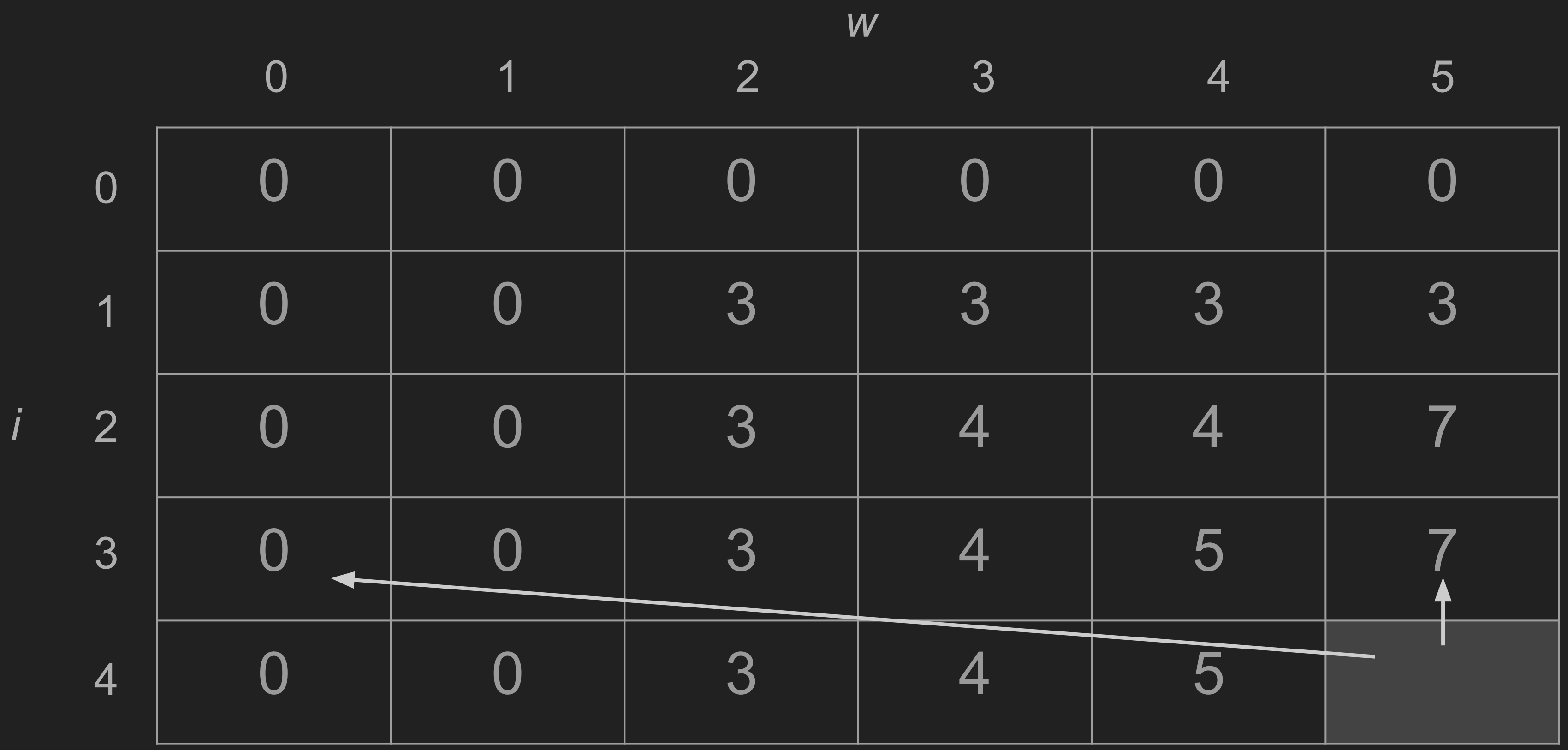
DP Example: Knapsack Problem

Items (*v*, *w*): {(3, 2), (4, 3), (5, 4), (6, 5)}

		w					
		0	1	2	3	4	5
i	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
	2	0	0	3	4	4	7
	3	0	0	3	4	5	7
	4	0	0	3	4	5	

DP Example: Knapsack Problem

Items (v, w): {(3, 2), (4, 3), (5, 4), (6, 5)}



	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

DP Example: Knapsack Problem

Items (*v*, *w*): {(3, 2), (4, 3), (5, 4), (6, 5)}

					w		
		0	1	2	3	4	5
	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
i	2	0	0	3	4	4	7
	3	0	0	3	4	5	7
	4	0	0	3	4	5	7

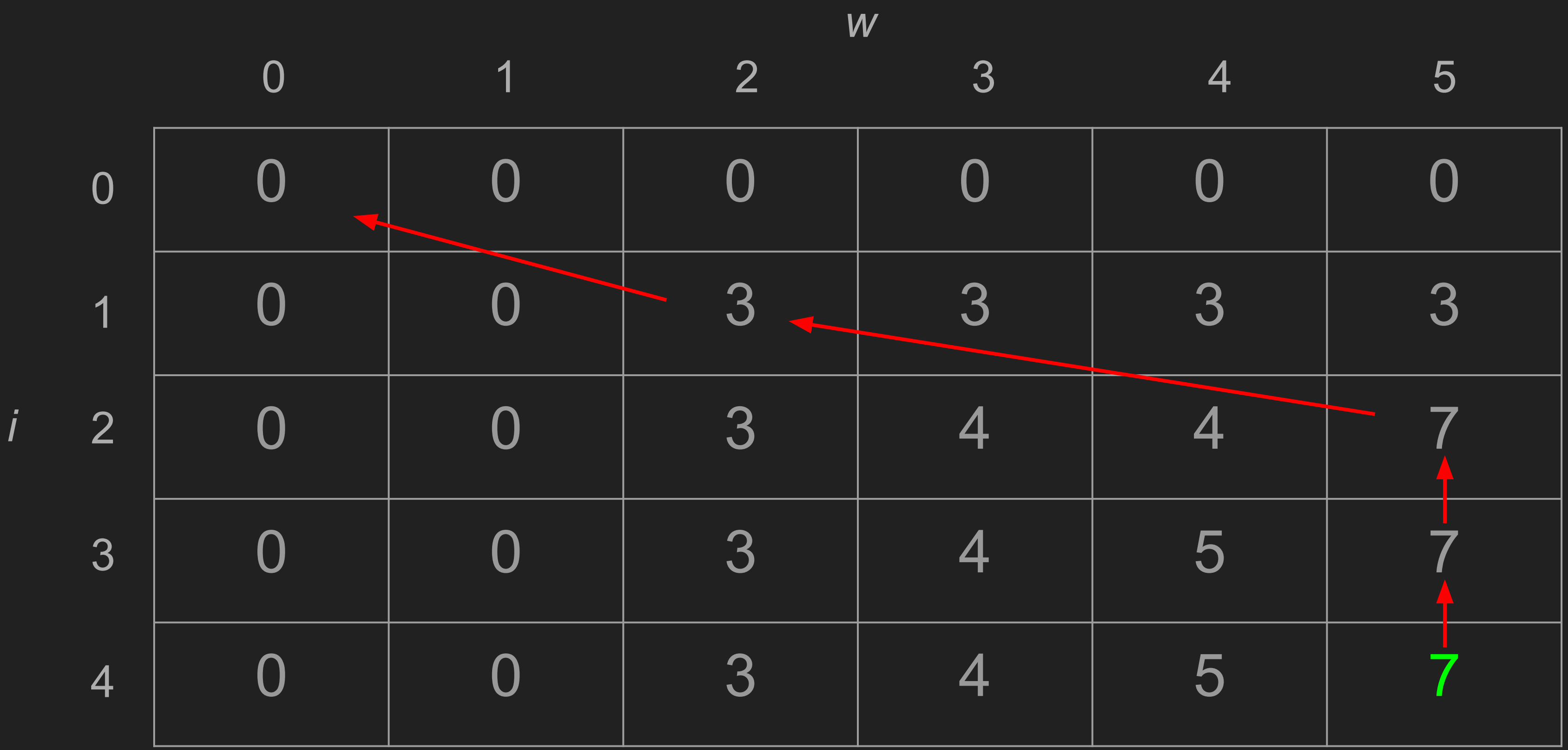
DP Example: Knapsack Problem

Items (*v*, *w*): {(3, 2), (4, 3), (5, 4), (6, 5)}

		w					
		0	1	2	3	4	5
i	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
	2	0	0	3	4	4	7
	3	0	0	3	4	5	7
	4	0	0	3	4	5	7

DP Example: Knapsack Problem

Items (*v*, *w*): {(3, 2), (4, 3), (5, 4), (6, 5)}



w

		0	1	2	3	4	5
	0	0	0	0	0	0	0
	1	0	0	3	3	3	3
i	2	0	0	3	4	4	7
	3	0	0	3	4	5	7
	4	0	0	3	4	5	7

DP Example: Knapsack Problem Implementation

```
def knapsack(items, max_weight):
    v = [x for (x, y) in items]
    w = [y for (x, y) in items]
    m = [[0 for i in range(max_weight + 1)] for j in
        range(len(items) + 1)]
    for i in range(max_weight + 1):
        m[0][i] = 0
    for i in range(items + 1):
        for j in range(max_weight + 1):
            if w[i] > j:
                m[i][j] = m[i-1][j]
            else:
                m[i][j] = max(m[i-1][j], m[i-1][j-w[i]] + v[i])
    return m[len(items)][max_weight]
```

DP Example: Knapsack Problem

- Runtime analysis: $O(nW)$
  - filling an n x W grid
  - constant time to fill a cell
- Space is also $O(nW)$
  - Only storing a single number
- Polynomial!
- ....except not really
- Runtime is proportional to W which isn't the size of the input but instead the magnitude of one of the input values.
  - 1 and 2^{63} can both be stored in the same amount of space
- $O(nW)$ is considered *pseudo-polynomial*
  - Technically, still exponential runtime
  - In practice, generally more useful than "genuine" exponential algorithms

DP Example: Knapsack Problem

- So, can we do better?
- Nope!
  - Or at least, if you figure out how you'll win $1 Million
  - ...and possibly also be able to break RSA
- Surprise: the Knapsack Problem is *NP-complete*
- Problems in NP are ones that are hard to solve, but it's easy to verify the solution
- NP-complete means that it is "equivalent" to other
  - Graph coloring
  - Traveling salesman
  - 3-SAT
  - etc. etc.
- Just because they aren't in P doesn't mean there aren't sometimes "good enough" algorithms