Assignment 4

Solution

Task 1 The interval scheduling problem

The following task will be based on the interval scheduling problem as explained in the book:

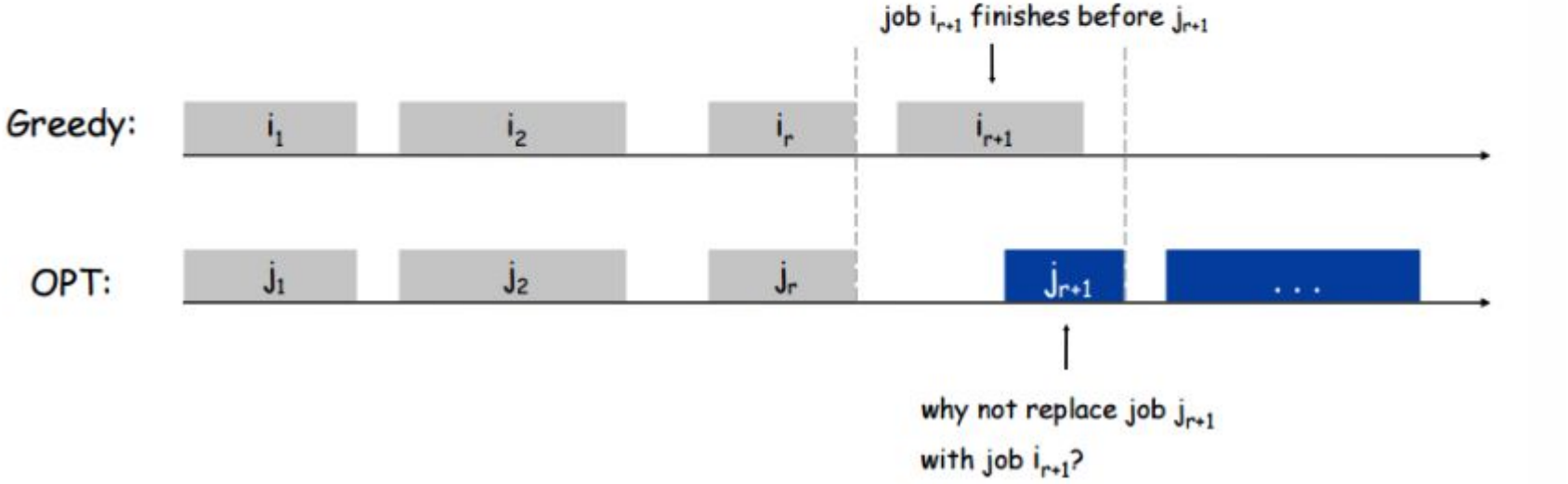
```
Initially let R be the set of all requests, and let A be empty
While R is not yet empty
    Choose a request i from R that has the smallest finishing time
    Add request i to A
    Delete all requests from R that are not compatible with request
    i
EndWhile
Return the set A as the set of accepted requests
```

a) Explain in your own words how the algorithm described above returns an optimal set

Solution:
Some version of the proof 4.3 in Algorithm Design is accepted here, as long as it isn't just directly copied.

It's not obvious that this algorithm produces the optimal solution. It does, though. How do we prove this? By contradiction.

- Let $i_1, i_2, \ldots i_k$ denote a set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote a set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



b) Given the following table of activities:

Activity	U	V	W	X	Y	Z
Start time	3	2	5	5	10	1
Finishing time	6	4	8	9	12	5

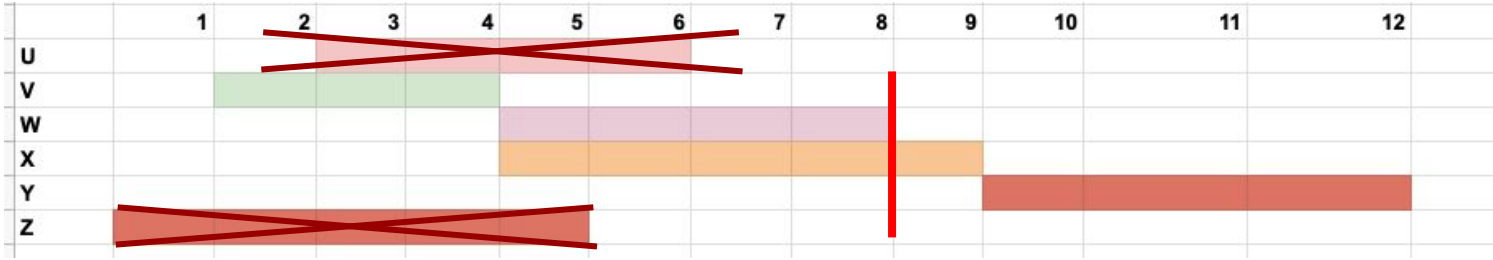
After running the interval scheduling algorithm described above, which of the intervals will be number two in the return set A?

Solution:
If we go through the steps of the algorithm we need to select the request with the smallest finish time, which is V. Then we select the next set, which is W, then we select Y.

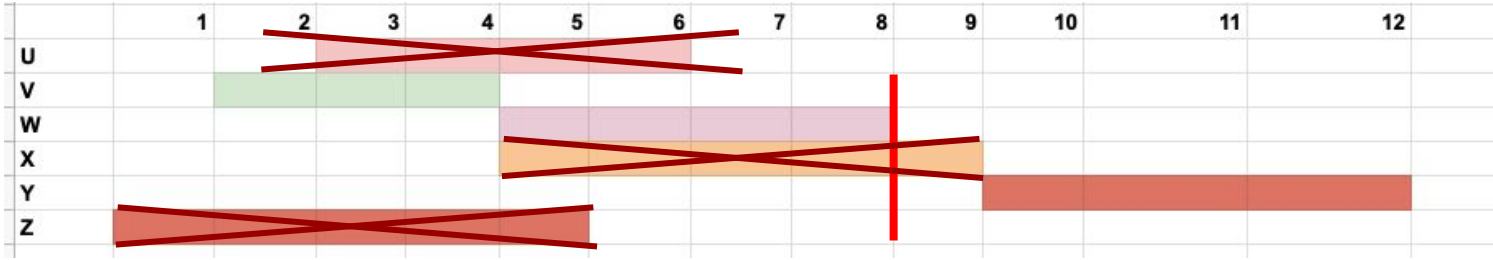
Meaning the interval W will be number two in the return set A


1
2
3
4
5
6
7
8
9
10
11
12
U
V
W
X
Y
Z

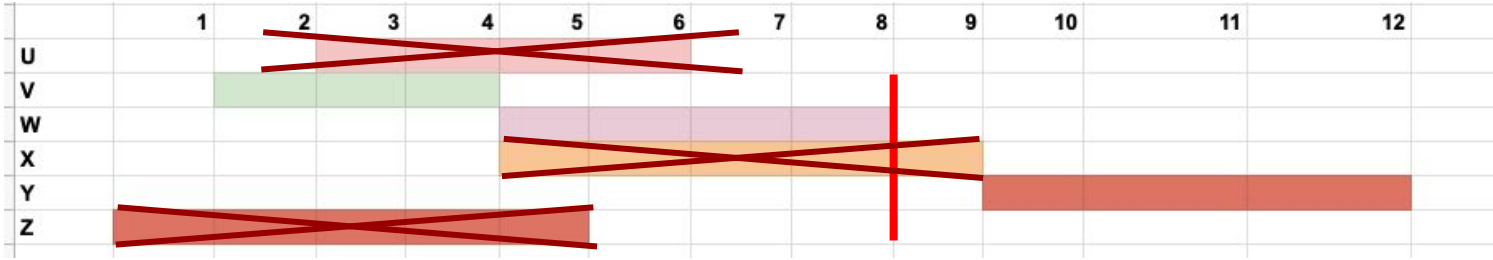
Earliest finish: V
{V}



Incompatible with Z and U
Earliest finish: W
{V, W}



Incompatible with X
Earliest finish: Y
{V, W, Y}



Incompatible with X
Earliest finish: Y
{V, W, Y}

And what will the size of A be?

Solution:
As we saw from above the size of A will be 3

Task 2 Minimum spanning trees

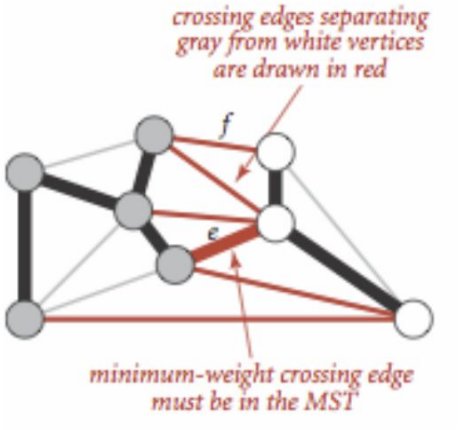
a) We have an edge (u, v) that has a strictly lower weight than all the other edges in a connected graph. Will this edge be included in the minimum spanning tree? Explain why or why not.

Solution:
If we have a minimum spanning tree T where (u, v) is not included, inserting (u, v) will make a cycle. Then we can remove another edge from the cycle, which we know has a higher value than (u, v). Doing this leads to a spanning tree with lower weight. Meaning that (u, v) has to be included in the spanning tree.

Task 2 Minimum spanning trees

a) We have an edge (u, v) that has a strictly lower weight than all the other edges in a connected graph. Will this edge be included in the minimum spanning tree? Explain why or why not.

Solution:
If we have a minimum spanning tree T where (u, v) is not included, inserting (u, v) will make a cycle. Then we can remove another edge from the cycle, which we know has a higher value than (u, v). Doing this leads to a spanning tree with lower weight. Meaning that (u, v) has to be included in the spanning tree.



b) Consider the following statement about Minimum Spanning Trees

We have a weighted undirected graph $G = (V, E)$ where all of the edges are weighted differently:

If the nodes V can be divided into two disjoint sets X and Y , then the minimum spanning tree will include the edge between X and Y with the lowest weight.

Is this correct? Explain why or why not.

Solution:
Yes, let's say we make a spanning tree without including the lightest edge between X and Y, then we could replace this edge with the lightest edge. Meaning a minimum spanning tree without the lightest edge between X and Y does not exist.

This problem is very similar to the one above, both test if the student understands the cut property.

b) Consider the following statement about Minimum Spanning Trees

We have a weighted undirected graph $G = (V, E)$ where all of the edges are weighted differently:

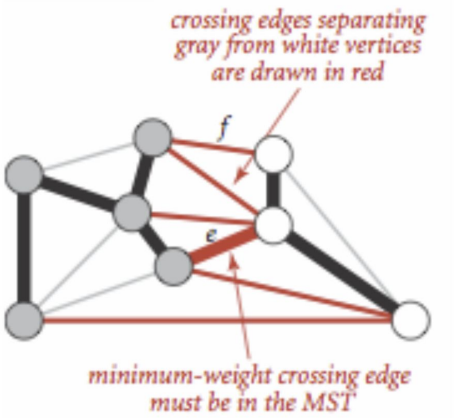
If the nodes V can be divided into two disjoint sets X and Y , then the minimum spanning tree will include the edge between X and Y with the lowest weight.

Is this correct? Explain why or why not.

Solution:

Yes, let's say we make a spanning tree without including the lightest edge between X and Y, then we could replace this edge with the lightest edge. Meaning a minimum spanning tree without the lightest edge between X and Y does not exist.

This problem is very similar to the one above, both test if the student understands the cut property.



Task 3 Shortest path in a graph

a) Can we use Dijkstra's algorithm to solve the shortest path problem on an undirected graph? Explain why or why not.

Solution:
Yes, although Dijkstra's algorithm is meant to be used on a directed graph, we can simply make an undirected graph directed by adding edges in each direction between every node that has an edge.

Task 3 Shortest path in a graph

a) Can we use Dijkstra's algorithm to solve the shortest path problem on an undirected graph? Explain why or why not.

Solution:
Yes, although Dijkstra's algorithm is meant to be used on a directed graph, we can simply make an undirected graph directed by adding edges in each direction between every node that has an edge.

An *undirected* graph is basically the same as a *directed* graph with **bidirectional** connections (= two connections in opposite directions) between the connected nodes.

So you don't really have to do anything to make it work for an undirected graph. You only need to know all of the nodes that can be reached from every given node through e.g. an **adjacency list**.

b) Explain, in your own words, why Dijsktra's algorithm works

Solution:
Some version of the proof 4.14 in Algorithm Design is accepted here, as long as it isn't just directly copied.

Task 4 Video game design

A video game developer has approached you since they heard rumors that you are taking the course TDT4121, and you know a thing or two about algorithms. They are making a side-scrolling video game where the main character moves from left to right, and he wants to add checkpoints to the game using an algorithm.

You can imagine the game as a flat straight line going from left to right, with obstacles placed along the way with varying distances between them. You want to add checkpoints so that each obstacle is within 400 pixels of a checkpoint, and you also want to add as few checkpoints as possible. Construct an efficient algorithm that places as few checkpoints as possible. Prove the correctness of your algorithm.

Solution:
We can (rather surprisingly) solve this with a greedy algorithm. If o is the left-most obstacle, we can place a checkpoint 400 pixels to the right of said obstacle. Now we remove all obstacles that are within the range of the checkpoint and repeat.

Let's think of the optimal solution to this problem. We know that from our solution we cover every obstacle possible within the range of the checkpoint since we move from the left. We can prove that this and the optimal set have the same size since the optimal set has to be at most as far to the right as our solution. Moving more to the right would not include the first obstacle. Let's say that the optimal solution is more to the left than our solution, this would not be optimal if there is some obstacle that is in range in our solution, but out of range if you move the checkpoint. Meaning that the optimal solution IS our solution.

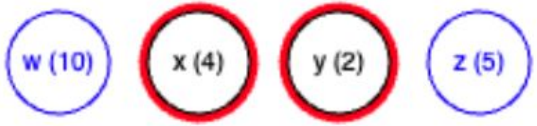
Task 5 Huffman coding

a) You are given a text file with the following frequencies:

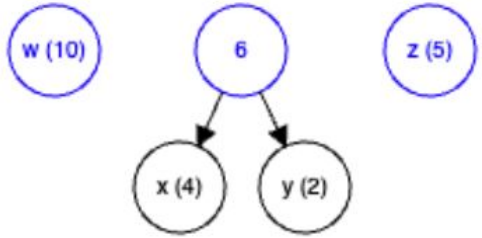
Letter	W	X	Y	Z
Frequency	10	4	2	5

What will the Huffman code for Y be in this example? Show your steps

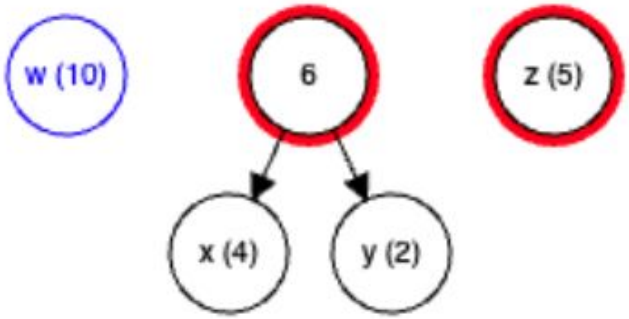
Removing two minimum elements from the priority queue.



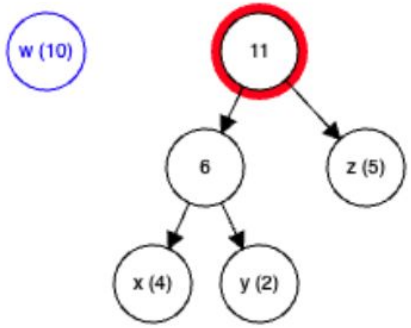
Reinserting the new root node in the priority queue.



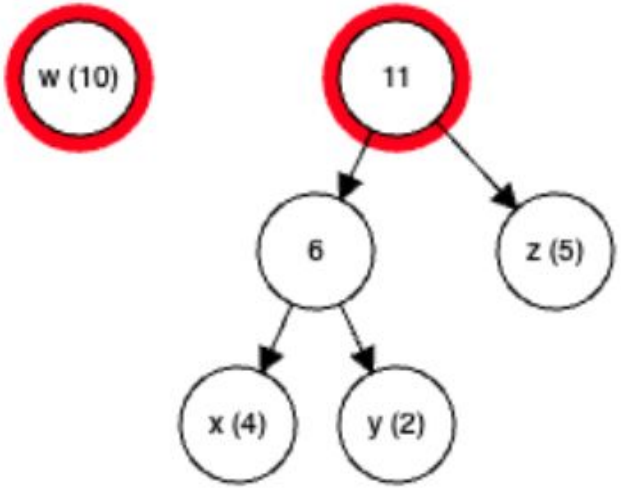
Removing two minimum elements from the priority queue.



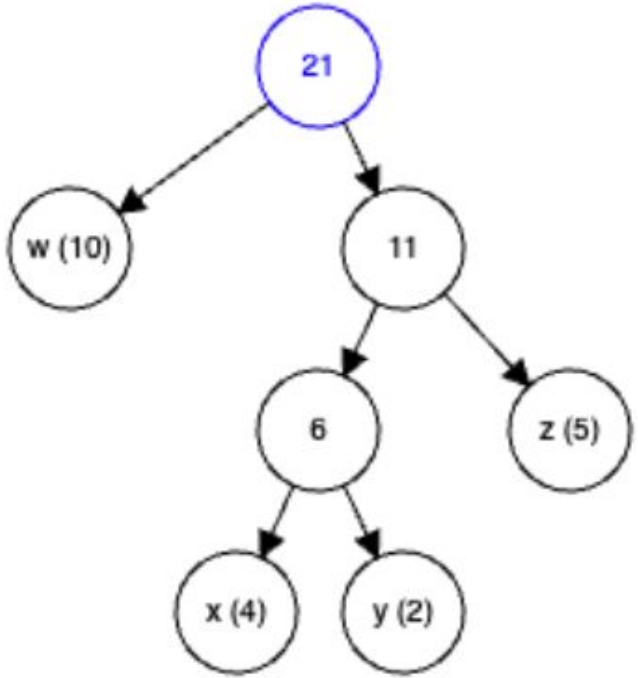
Reinserting the new root node in the priority queue.



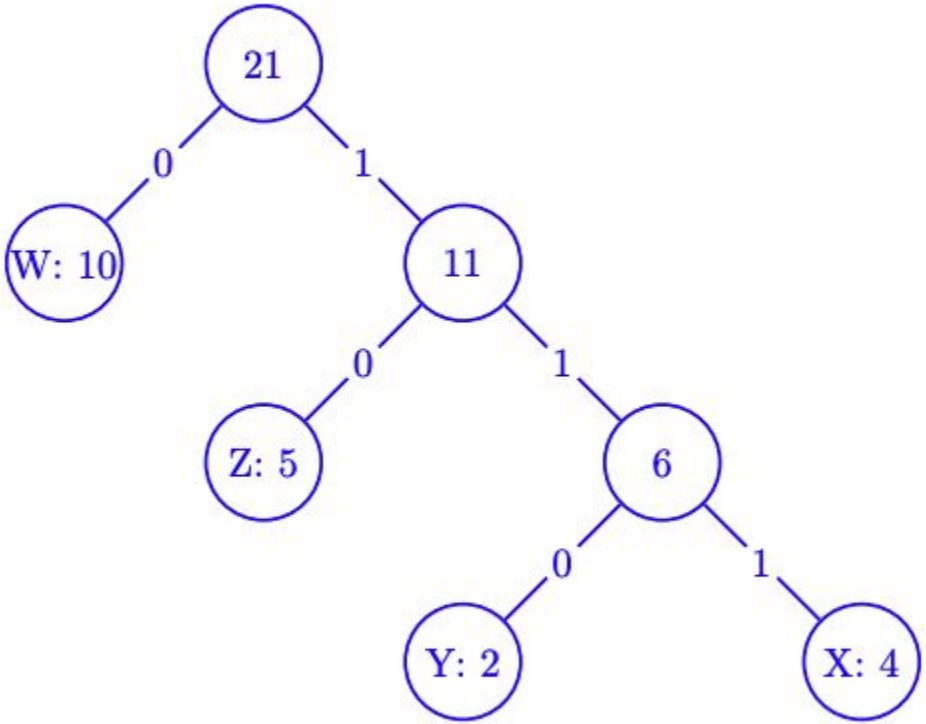
Removing two minimum elements from the priority queue.



Reinserting the new root node in the priority queue.

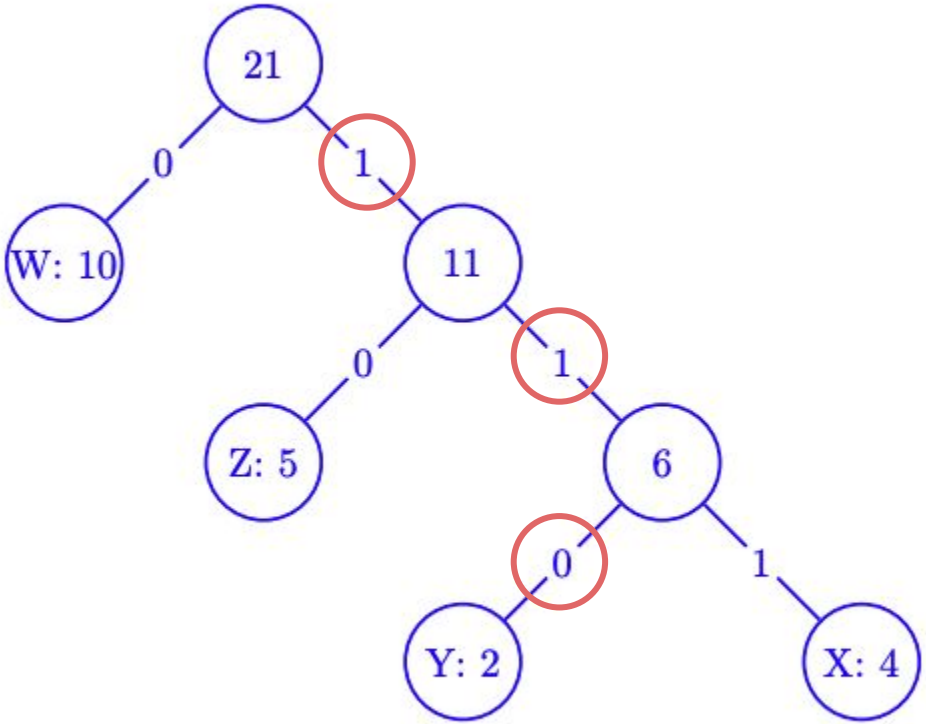


Solution:
We start by drawing a Huffman-tree based on the frequency table:



We can then traverse down the tree where we find that the code for Y is ‘110‘

Solution:
We start by drawing a Huffman-tree based on the frequency table:



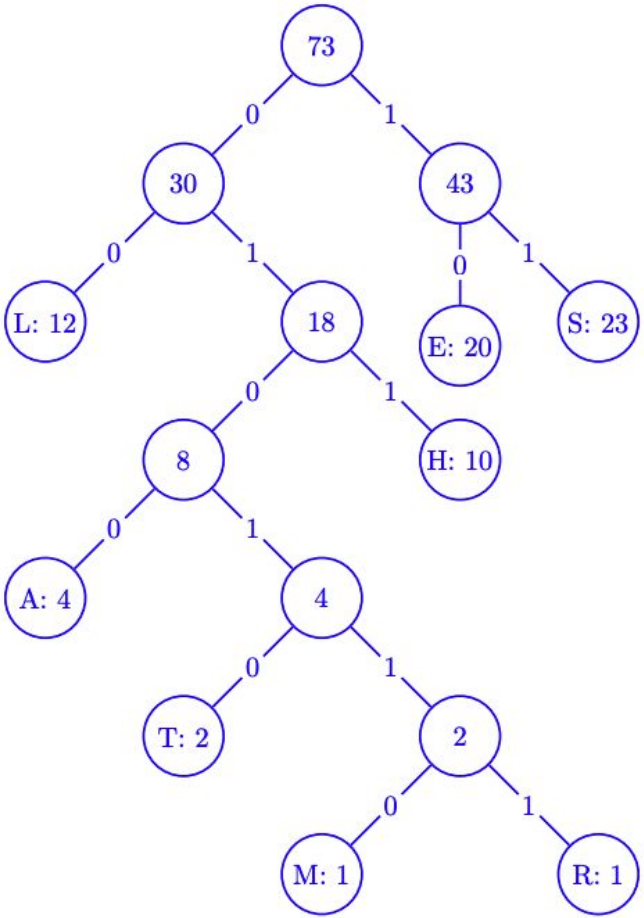
We can then traverse down the tree where we find that the code for Y is ‘110‘

b) Provide a Huffman encoding for the phrase: "SHE SELLS SEASHELLS SHE SEES SEASHELLS THE SHELLS SHE SELLS ARE SEASHELLS SHE SEES THEM". Explain your answer by making a frequency table and drawing a Huffman tree, and then creating the codes based on the tree. You can ignore spaces.

Solution:
Let's start by creating a frequency table:

Letter	S	H	E	A	T	M	R	L
Frequency	23	10	29	4	2	1	1	12

Now let's draw the Huffman tree for this frequency table. Keep in mind that the Huffman algorithm is non-deterministic, so your solution could differ from this as this sequence of numbers will lead to some equal values, so nodes could differ from being on the left or right side of its parent.



e graph to create our Huffman codes:

Letter	S	H	E	A	T	M	R	L
Code	11	011	10	0100	01010	010110	0101111	00

Task 6 ★ *Optional*

a) In a Huffman-code for an alphabet with $n \geq 1$ symbols, what is the largest length a codeword for a symbol can have? Provide proof of your statement.

Solution:
Answering $n - 1$ is not sufficient, as this is not true for n $= 1$, a more detailed explanation is required to get credit

Since the Huffman codes are stored in a binary tree and the letter with the lowest frequency is stored at the bottom, then the largest lenght of a symbol will be $n - 1$ and the shortest length would be , this however does not hold true if there are $n = 1$ symbols. Then the longest length would be 1.

Part 2: Programming

```python
class HeapNode:
    def __init__(self, character, value):
        self.character = character
        self.value = value
        self.left_child = None
        self.right_child = None

    def __lt__(self, other):
        if not isinstance(other, HeapNode):
            return False
        return self.value < other.value

    def __eq__(self, other):
        if not isinstance(other, HeapNode):
            return False
        return self.value == other.value

    def __str__(self) -> str:
        string = f"Character: '{self.character}' "
        string += f"Value: '{self.value}' "
        string += f"Left child: '{self.left_child}' "
        string += f"Right child: '{self.right_child}' "
        return string

    def __repr__(self) -> str:
        return self.__str__()
```

2 - Making the frequency heap

```
import heapq


def make_frequency_heap(string: str) -> list:
    # Initialize freq dictionary and heap array
    freq = {}
    heap = []
    # loop through the characters in the string
    for character in string:
        if character not in freq:
            # insert them into the freq dictionary,
            # with the character being they key and the value being the frequency
            freq[character] = 1
        else:
            # increase the frequency
            freq[character] += 1

    for character in freq:
        # insert the values from the freq dictionary into the heap
        node = HeapNode(character, freq[character])
        heapq.heappush(heap, node)

    return heap


teststring = "ABBCCCDDDD"
heap = make_frequency_heap(teststring)
for node in heap:
    print(node)
```

3 - Merging the codes

```
def merge_code(heap: list) -> HeapNode:
    # While there is more than one node in the heap
    while len(heap) > 1:
        # Extract the two nodes with the lowest frequency letters from the heap
        # (remember that the letter with the lowest frequency will always be at the top of the heap)
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)
        # Create a new node that has the sum of the values of the two nodes as its value,
        # and the two nodes as left and right child respectively
        newnode = HeapNode(None, node1.value + node2.value)
        newnode.left_child = node1
        newnode.right_child = node2
        # Push this new node into the heap
        heapq.heappush(heap, newnode)
    # Return the root of the tree
    return heap[0]
```

4 - Traversing the tree

```python
def traverse_huffman(rootnode: HeapNode) -> dict:
    # Store the codes for each letter
    codes = {}
    # Keeps track of the current code
    current_code = ""
    # traverse recursively
    traverse_huffman_recursive(rootnode, current_code, codes)
    # return finished encoding
    return codes


def traverse_huffman_recursive(node: HeapNode, current_code: str, codes: dict) -> None:
    # if there exists a character in the node,
    if node.character != None:
        # append current_code as the value and the character
        codes[node.character] = current_code
        # as the key in codes and return
        return

    # Left side of the tree, append 0 to the current node
    traverse_huffman_recursive(node.left_child, current_code + "0", codes)
    # Right side of the tree, append 1 to the current node
    traverse_huffman_recursive(node.right_child, current_code + "1", codes)
```

Running the program

Here we have a main function to run the whole program, use it to test if you get the correct output:

```
def main():
    text = "ABBBBCCCDDEEEEAAAEEBBBCC"
    heap = make_frequency_heap(text)
    rootnode = merge_code(heap)
    encoding = traverse_huffman(rootnode)
    print(encoding)


main()
```

```
{'C': '00', 'E': '01', 'D': '100', 'A': '101', 'B': '11'}
```

Dynamic Programming

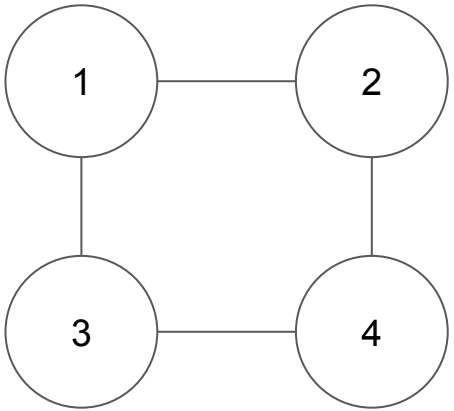
Main idea: reuse solutions to overlapping subproblems

- Some problems have overlapping subproblems. Instead of solving the the subproblem again and again, can we store the solution after the first time we solve it and retrieve it every time we encounter the same problem again?

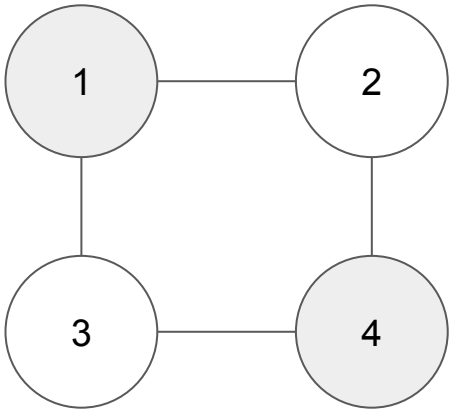
When can we use dynamic programming?

1. **Optimal substructure**: the optimal solution to the main problem builds on the optimal solutions of its subproblems
2. **Overlapping subproblems**: subproblems occur more than once

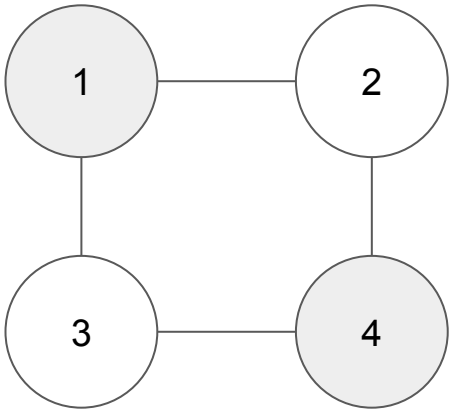
Example: finding the shortest path



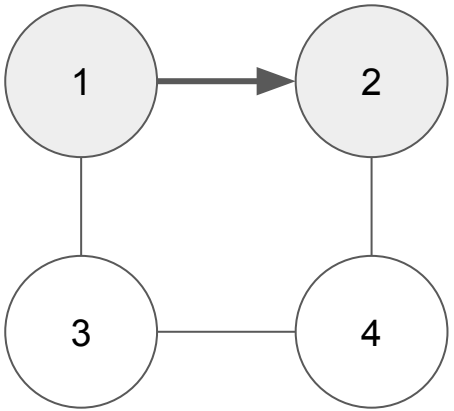
Want to go from 1 to 4

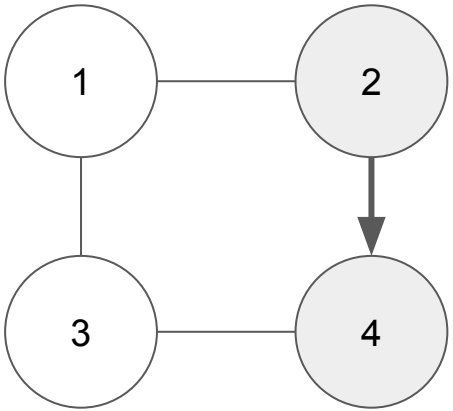


Can we decompose this into a subproblem?

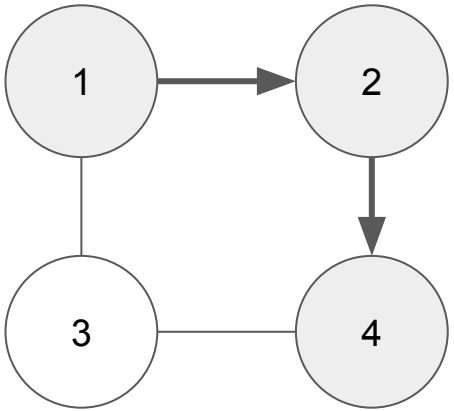


Shortest path from 1 -> 2 and from 2 -> 4

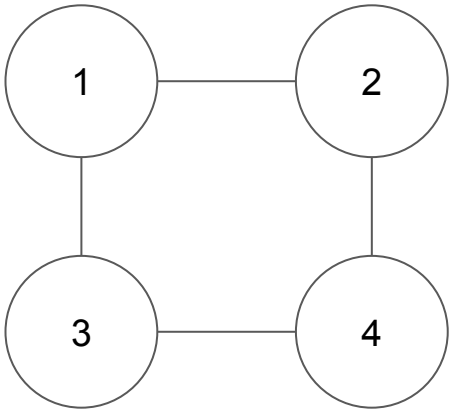




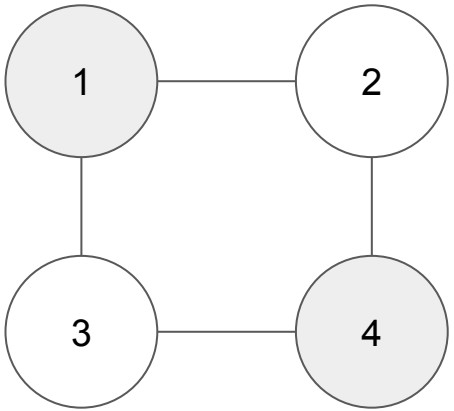
Combining these gives us the shortest path from 1 -> 4



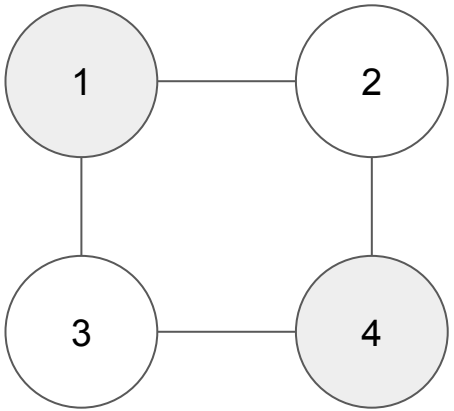
Example: finding the **longest** path



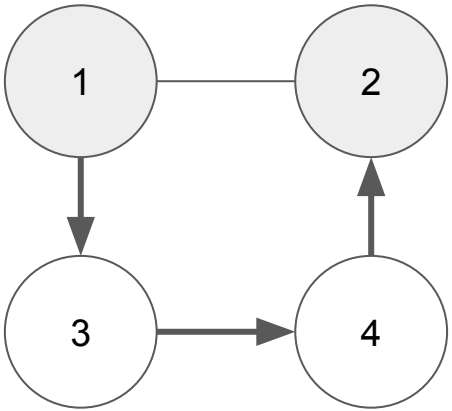
Want to go from 1 to 4

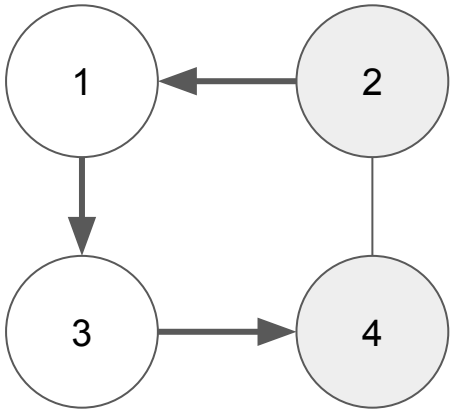


Can the longest path be decomposed into subproblems?

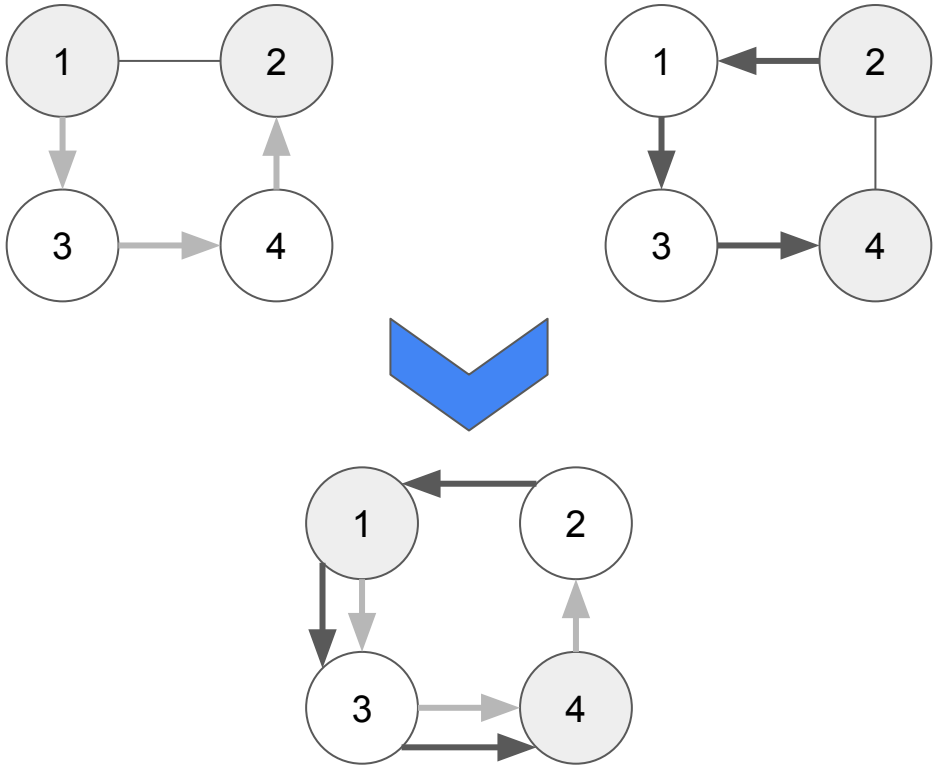


We try the same decomposition: 1->2 and 2->4





Does not work! We already went through 4

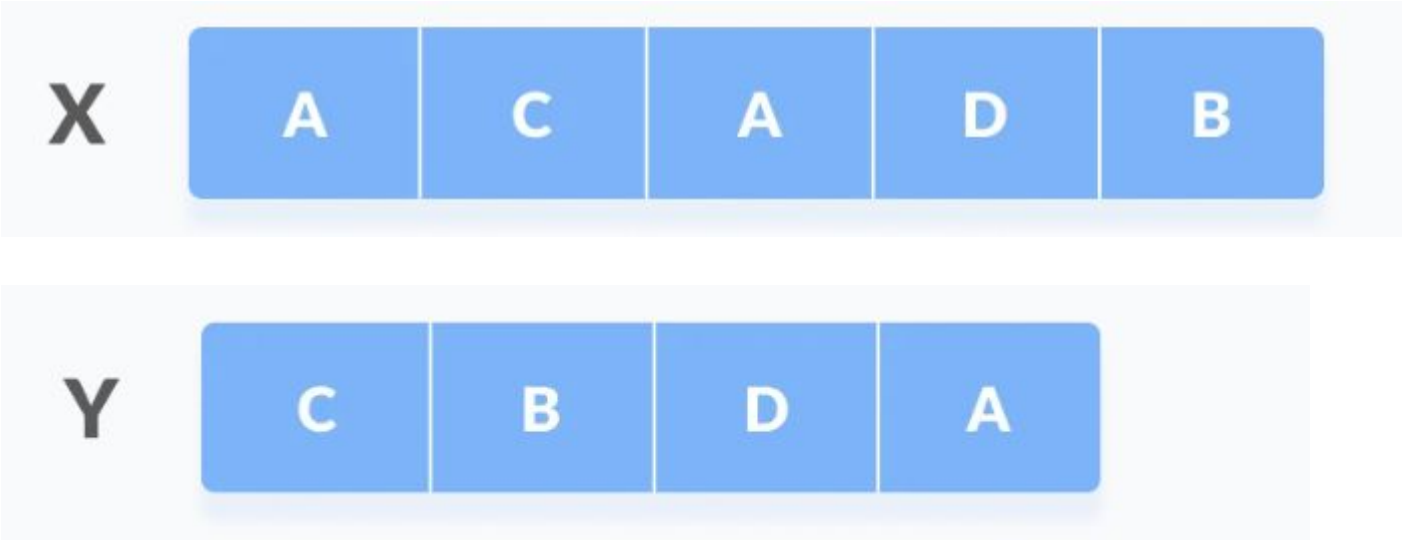


Shortest path has optimal substructure, longest path does **not**

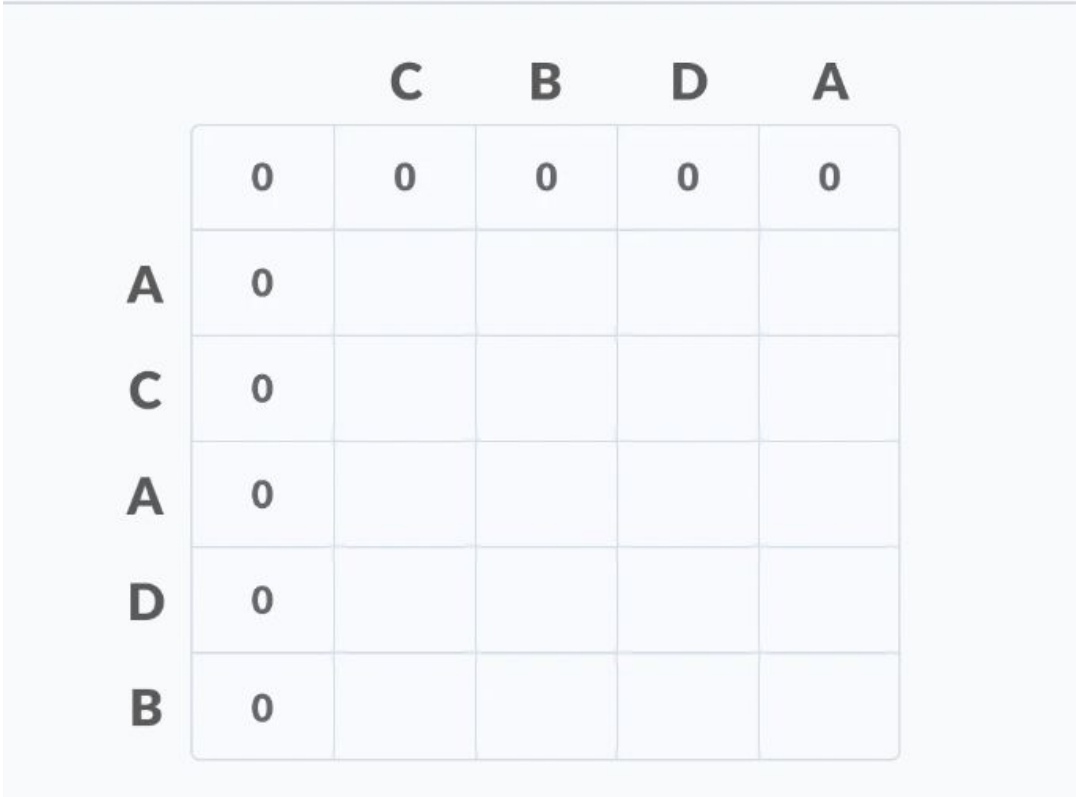
Example: LCS

Longest Common Subsequence

What is the longest common substring of two strings?



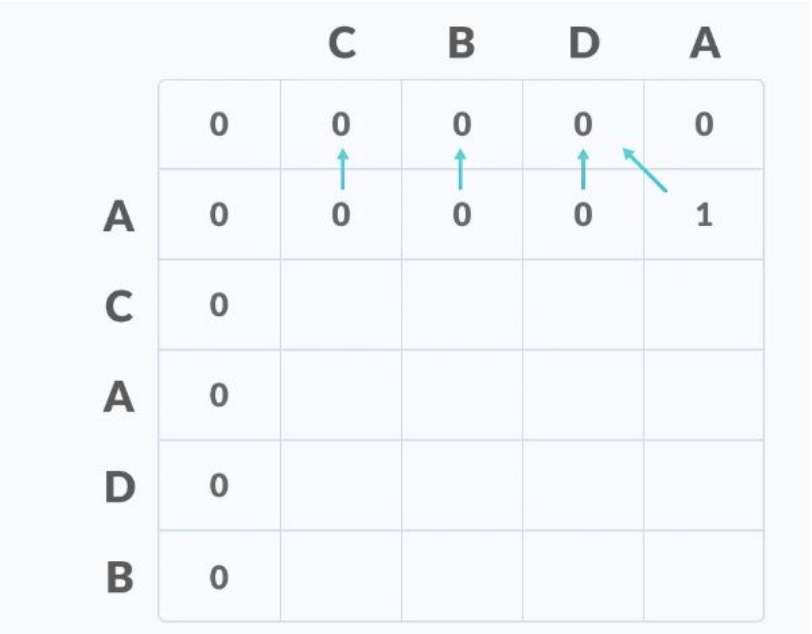
1. Create a table x*y



		C	B	D	A
	0	0	0	0	0
A	0				
C	0				
A	0				
D	0				
B	0				

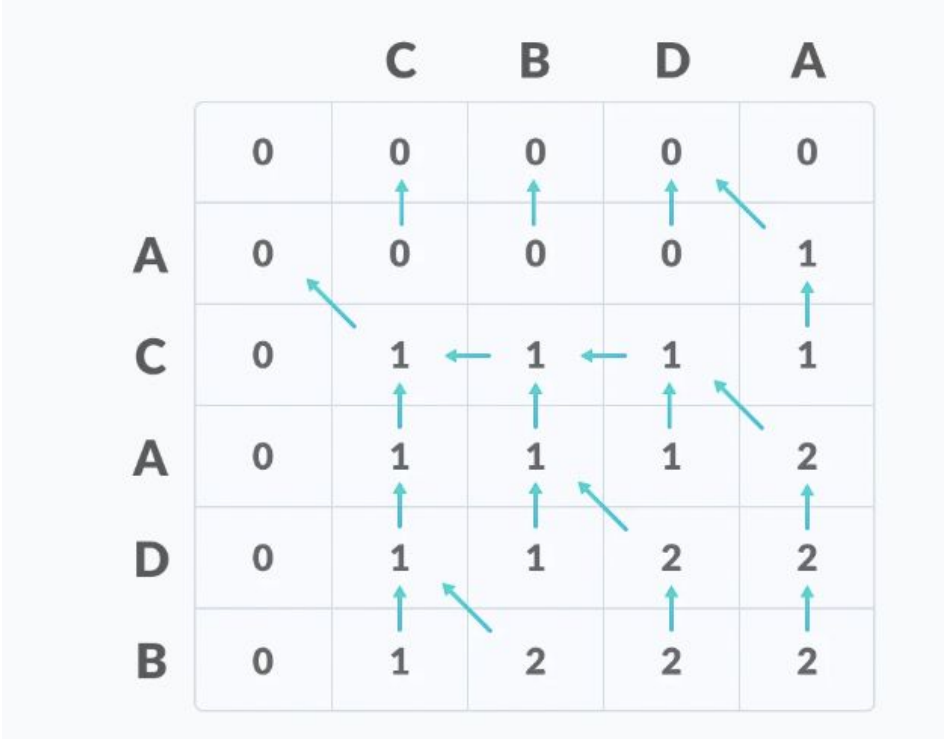
Fill the values using these rules:

1. If the letter on the current row and current column is the same, fill the current cell with **the value of the diagonal element + 1**
2. Else: take the maximum value from the previous column/row

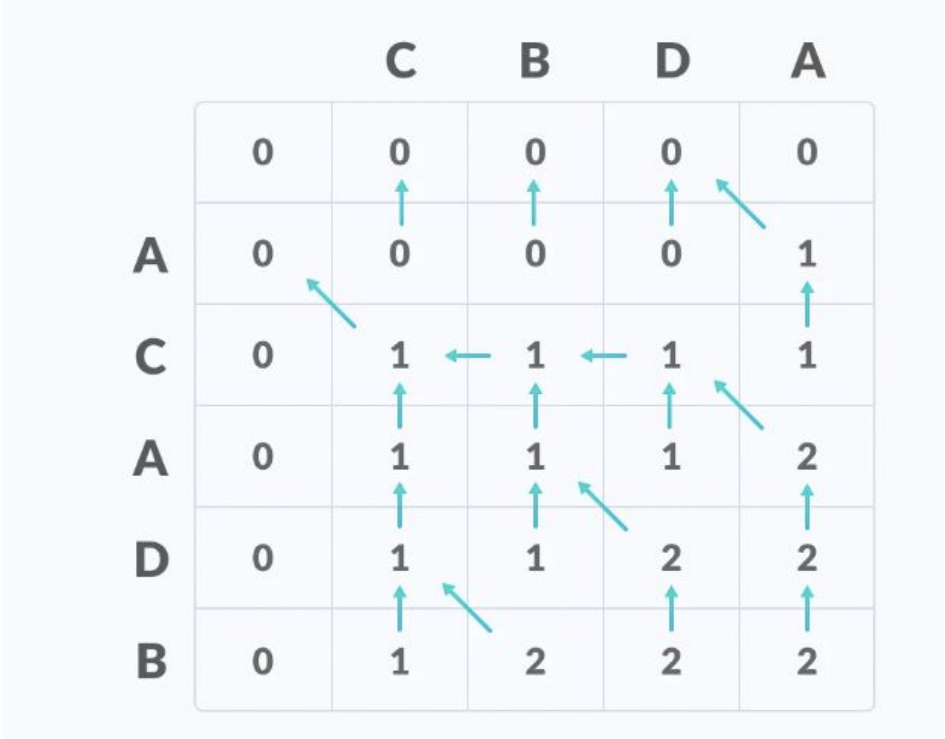


Fill the values using these rules:

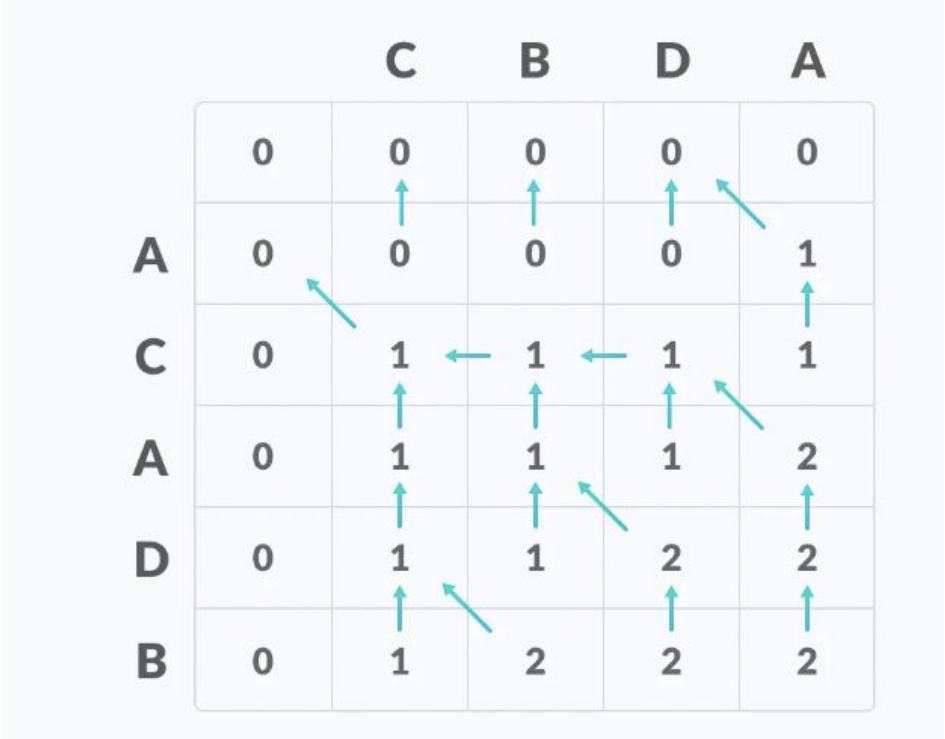
1. If the letter on the current row and current column is the same, fill the current cell with **the value of the diagonal element + 1**
2. Else: take the maximum value from the previous column/row
3. **Repeat steps 1 and 2 until the whole table is filled**



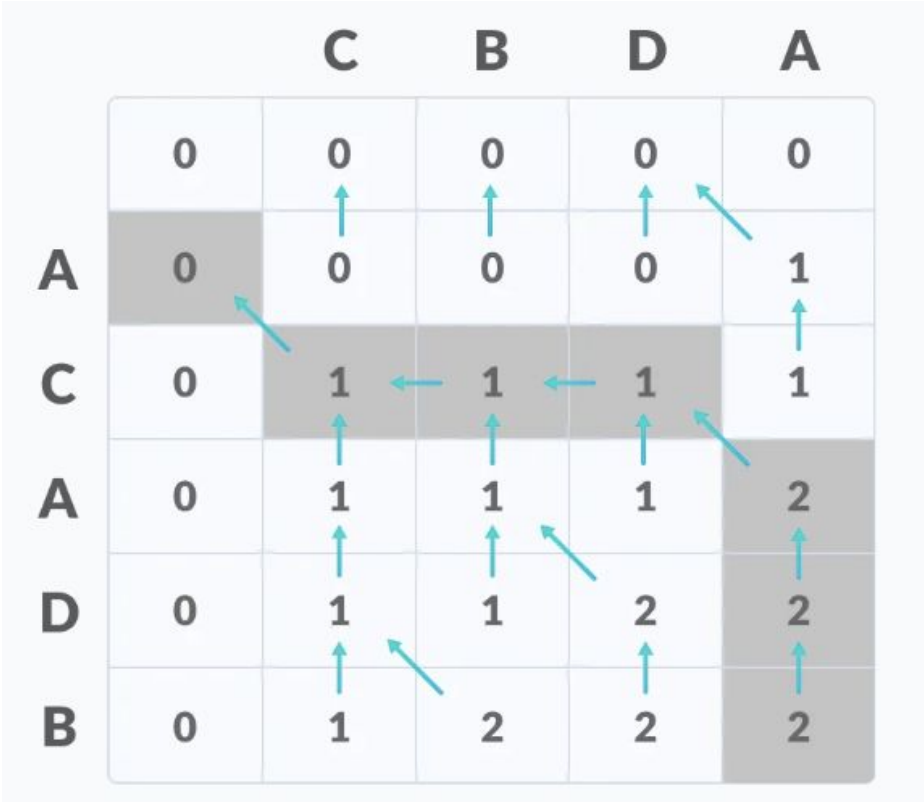
The value in the very last cell is the length of the LCS



How do we find the actual sequence (not just the length)?

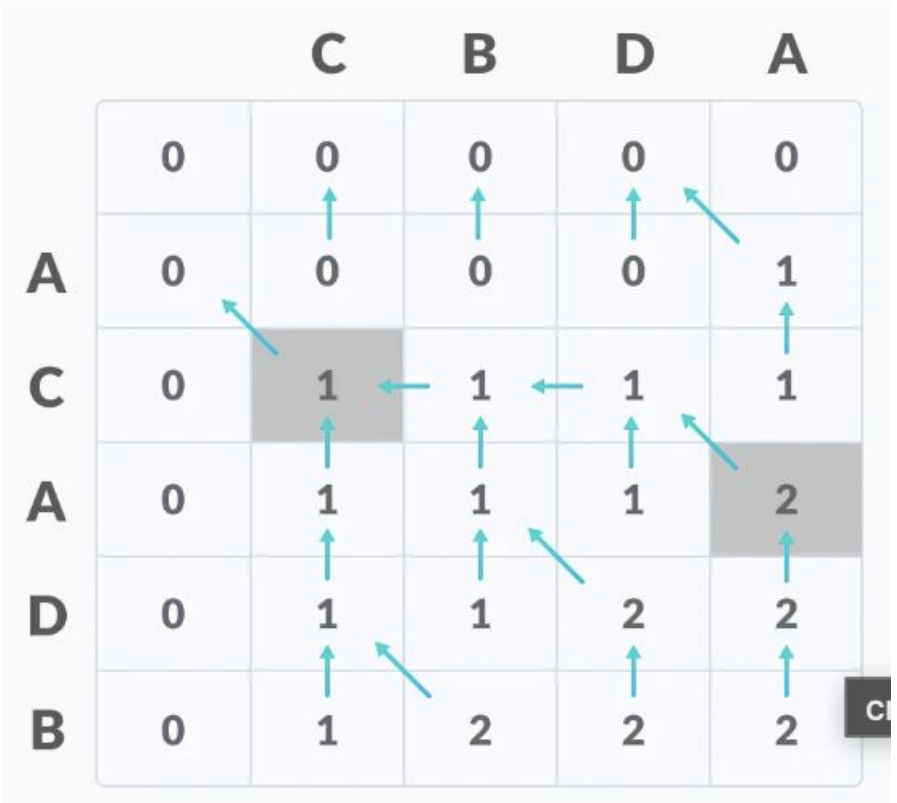


Save the arrows and backtrack



Remove all but diagonal arrows

This leaves us with CA, which is the LCS



How does this translate to code?

1. Let X and Y be two given strings
2. Initialize a table **LCS** of size X.length * Y.length

Fill the first row and column with 0s

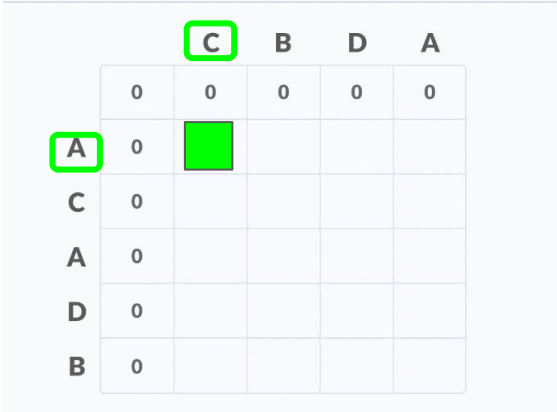
1. Let X and Y be two given strings
2. Initialize a table **LCS** of size X.length * Y.length
3. Fill the first row and column with 0s

Starting at (1, 1), go through every cell

1. Let X and Y be two given strings
2. Initialize a table **LCS** of size X.length * Y.length
3. Fill the first row and column with 0s
4. For every character X[i] in X[1:]:
   For every character Y[j] in Y[1:]:

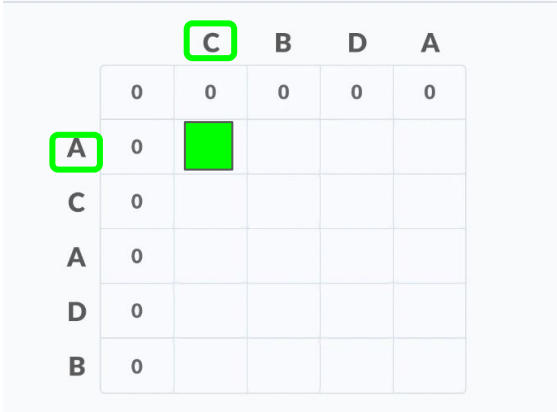
Check if the characters match

1. Let X and Y be two given strings
2. Initialize a table **LCS** of size X.length * Y.length
3. Fill the first row and column with 0s
4. For every character X[i] in X[1:]:
    For every character Y[j] in Y[1:]:
        If X[i] = Y[j]:
            LCS[i, j] = LCS[i-1, j-1] + 1



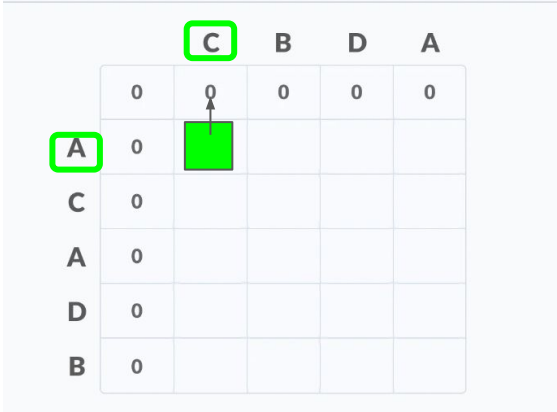
No match

1. Let X and Y be two given strings
2. Initialize a table **LCS** of size X.length * Y.length
3. Fill the first row and column with 0s
4. For every character X[i] in X[1:]:
    For every character Y[j] in Y[1:]:
        If X[i] = Y[j]:
            LCS[i, j] = LCS[i-1, j-1] + 1
            Point an arrow from LCS[i, j] to LCS[i-1, j-1]
    Else: LCS[i, j] = max(LCS[i-1, j], LCS[i, j-1])
        Point an arrow to max(LCS[i-1, j], LCS[i, j-1])



Max of (0, 0) is 0

1. Let X and Y be two given strings
2. Initialize a table **LCS** of size X.length * Y.length
3. Fill the first row and column with 0s
4. For every character X[i] in X[1:]:
    For every character Y[j] in Y[1:]:
        If X[i] = Y[j]:
            LCS[i, j] = LCS[i-1, j-1] + 1
            Point an arrow from LCS[i, j] to LCS[i-1, j-1]
    Else: LCS[i, j] = max(LCS[i-1, j], LCS[i, j-1])
        Point an arrow to max(LCS[i-1, j], LCS[i, j-1])



What is the runtime complexity of this algorithm?

1. Let X and Y be two given strings
2. Initialize a table **LCS** of size X.length * Y.length
3. Fill the first row and column with 0s
4. For every character X[i] in X: *# Loop runs X.length times*
   For every character Y[j] in Y: *# Loop runs Y.length times*
   If X[i] = Y[j]: *# This takes constant time*
   LCS[i, j] = LCS[i-1, j-1] + 1
   Point an arrow from LCS[i, j] to LCS[i-1, j-1]
   Else: LCS[i, j] = max(LCS[i-1, j], LCS[i, j-1])
   Point an arrow to max(LCS[i-1, j], LCS[i, j-1])

Runtime is O(X.length*Y.length)