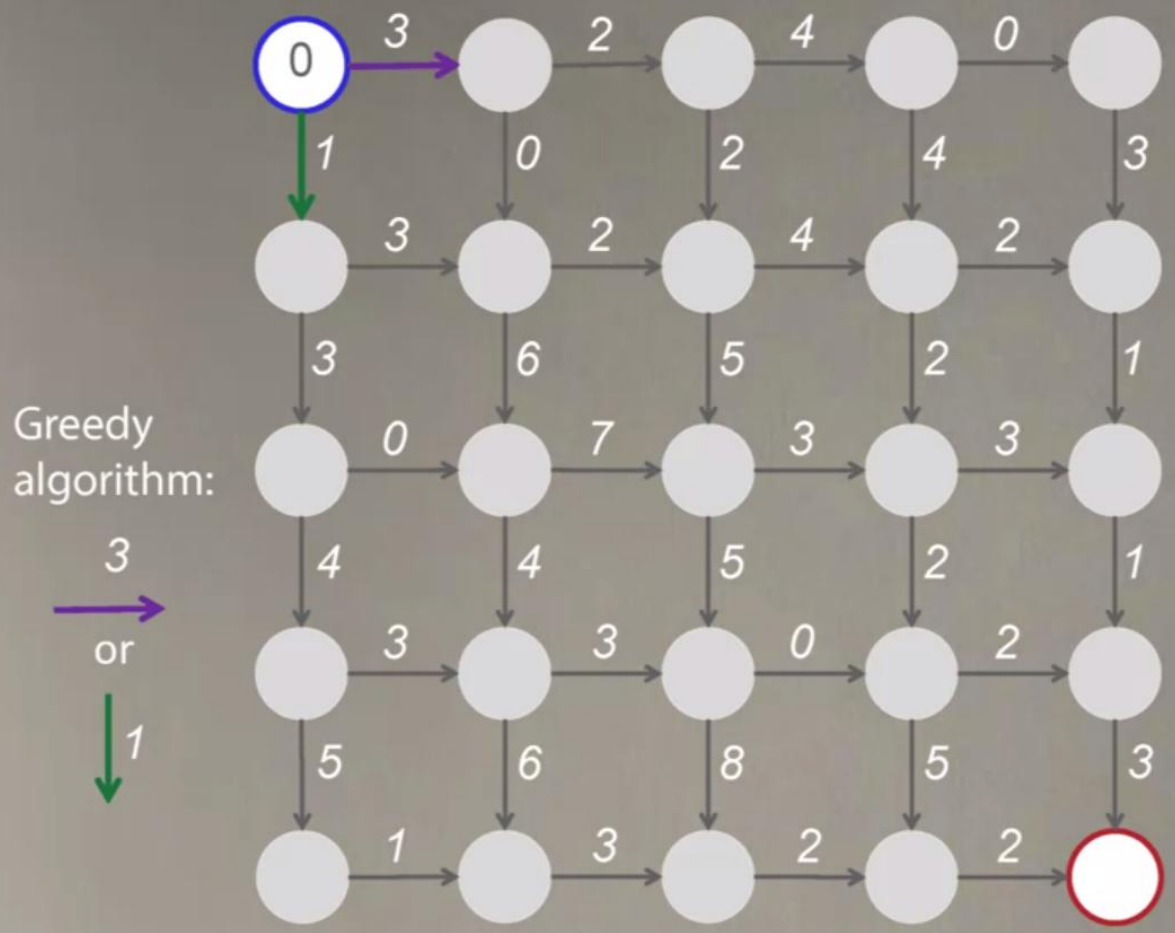
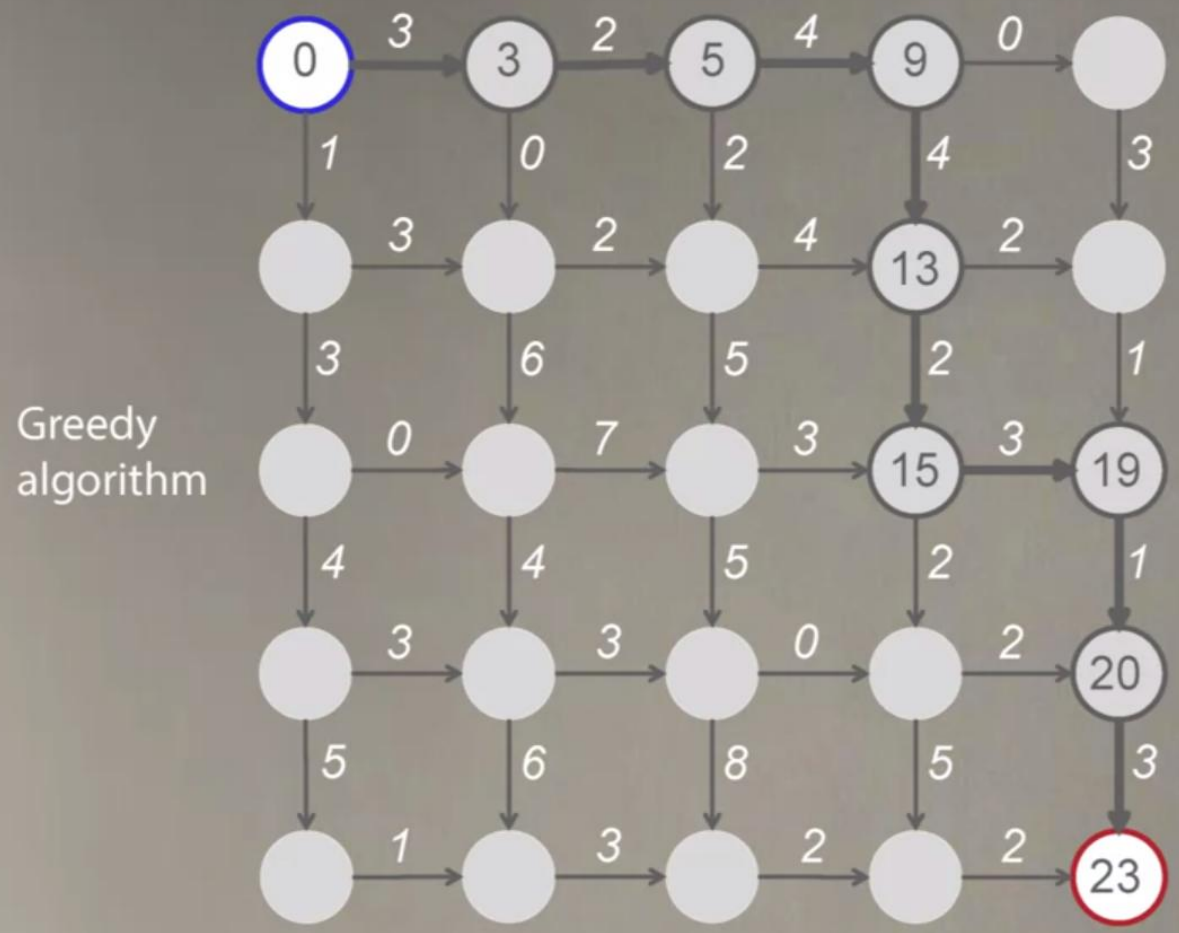
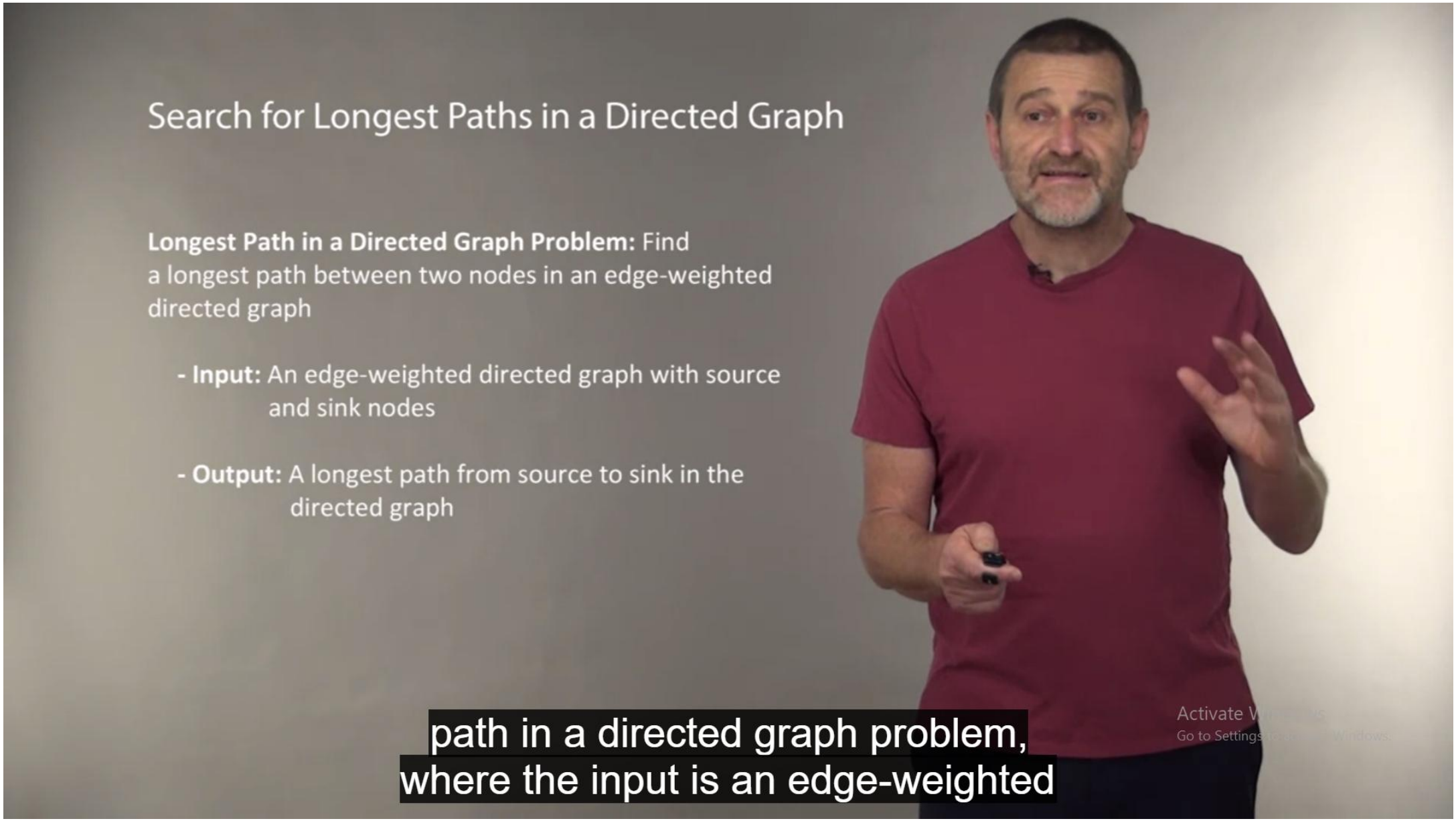
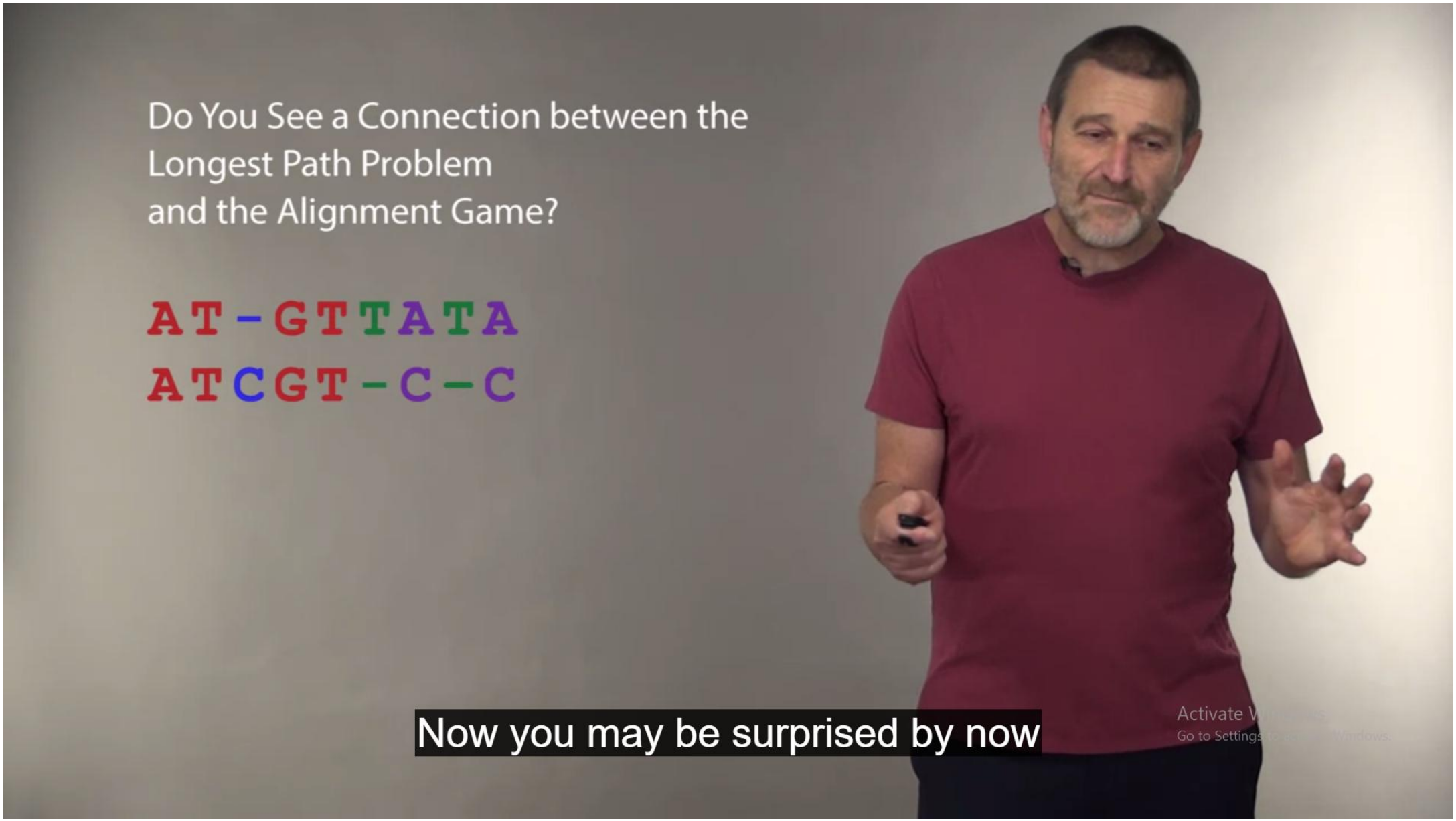


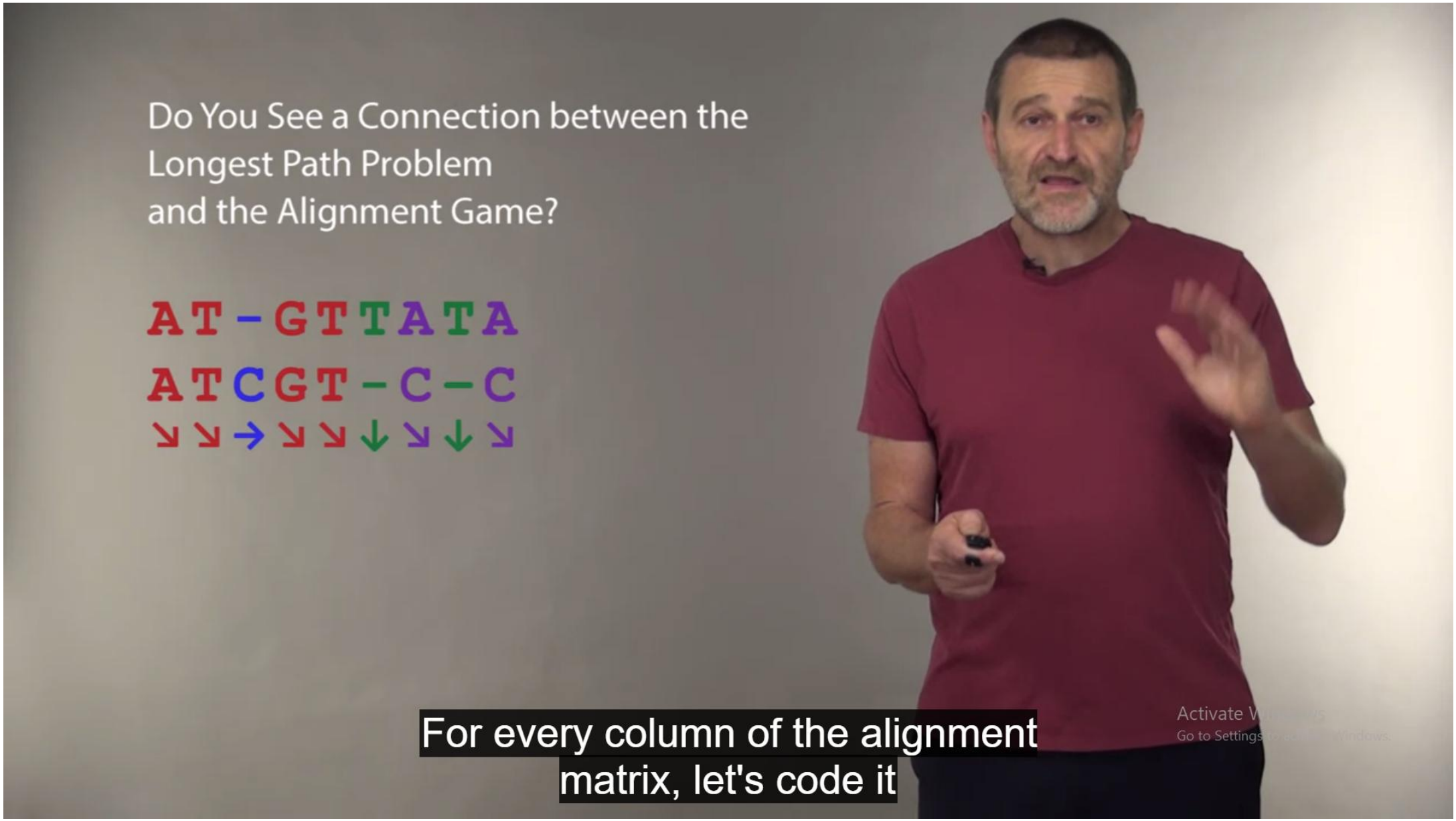
If you move east, we will visit three attractions immediately.



we arrive to the source, visiting 23 attractions.

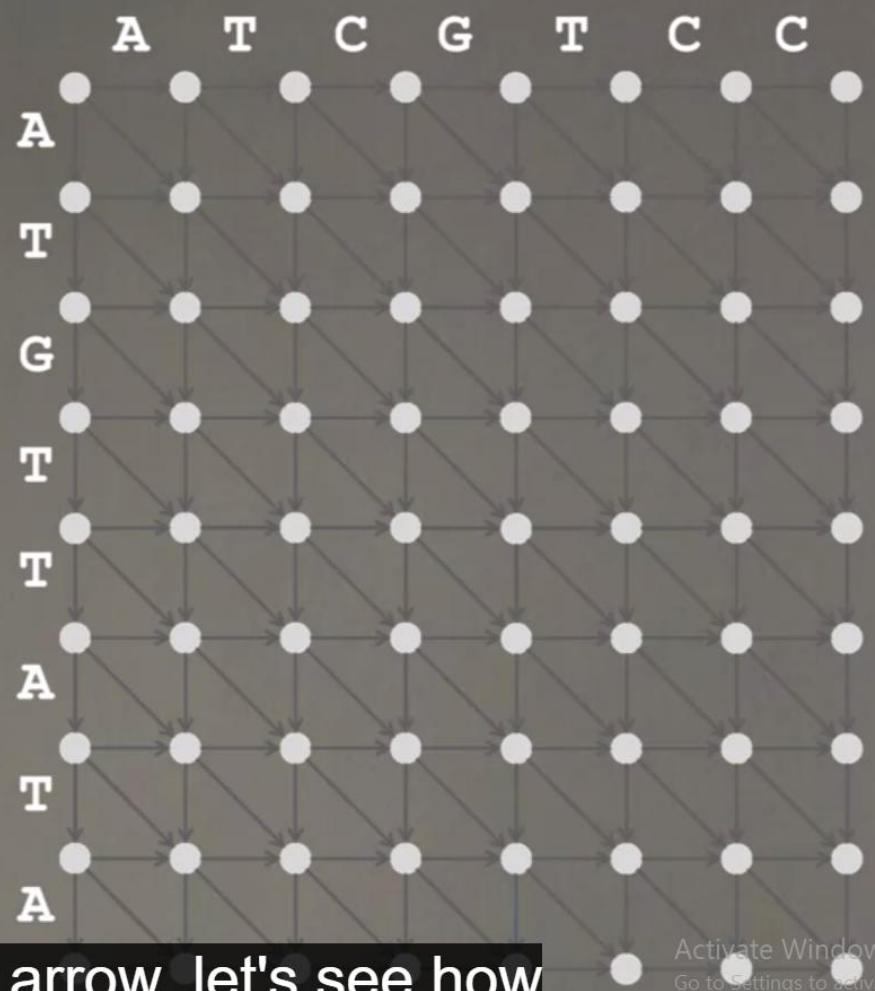

Search for Longest Paths in a Directed Graph
Longest Path in a Directed Graph Problem: Find a longest path between two nodes in an edge-weighted directed graph
- Input: An edge-weighted directed graph with source and sink nodes
- Output: A longest path from source to sink in the directed graph
path in a directed graph problem,
where the input is an edge-weighted


Do You See a Connection between the
Longest Path Problem
and the Alignment Game?
AT-GTTATA
ATCGT-C-C
Now you may be surprised by now


Do You See a Connection between the
Longest Path Problem
and the Alignment Game?
AT-GTTATA
ATCGT-C-C
↘ ↘ → ↘ ↘ ↓ ↘ ↓ ↘
For every column of the alignment
matrix, let's code it

alignment → path

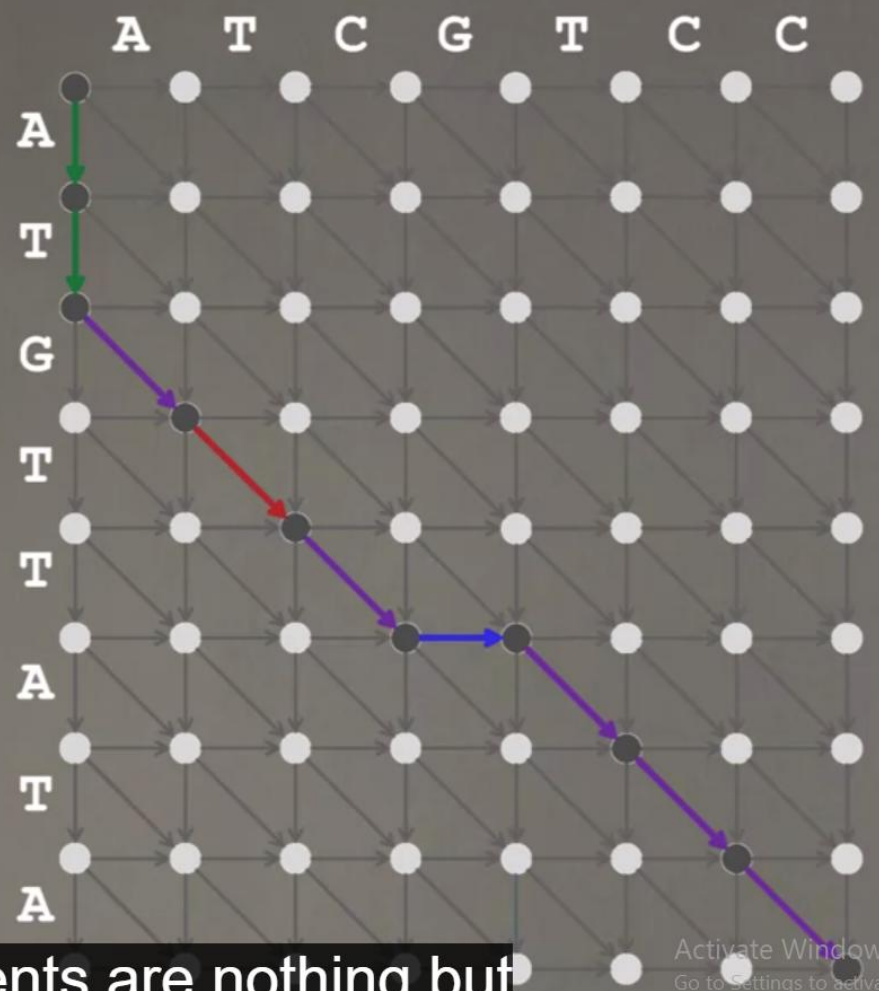
A T - G T T A T A
A T C G T - C - C
↘ ↘ → ↘ ↘ ↓ ↘ ↓ ↘



after we design this arrow, let's see how this arrow would translate

?
path → alignment

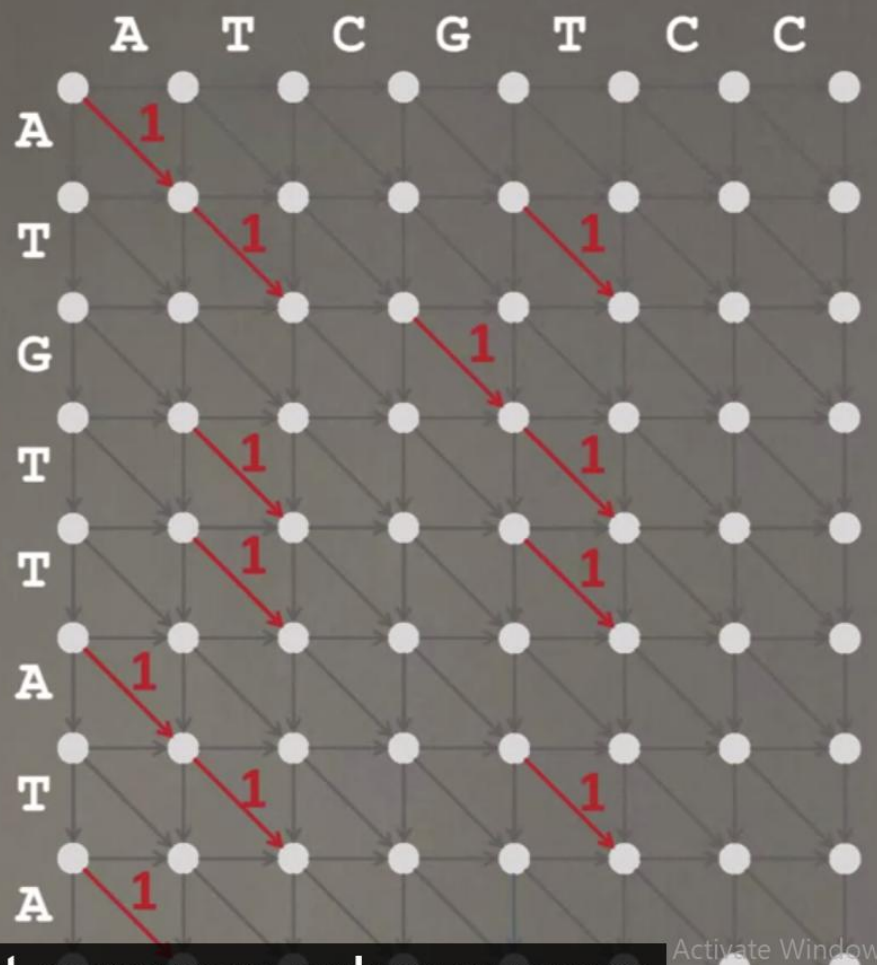
A	T	G	T	T	-	A	T	A
-	-	A	T	C	G	T	C	C
↓	↓	↘	↘	↘	→	↘	↘	↘



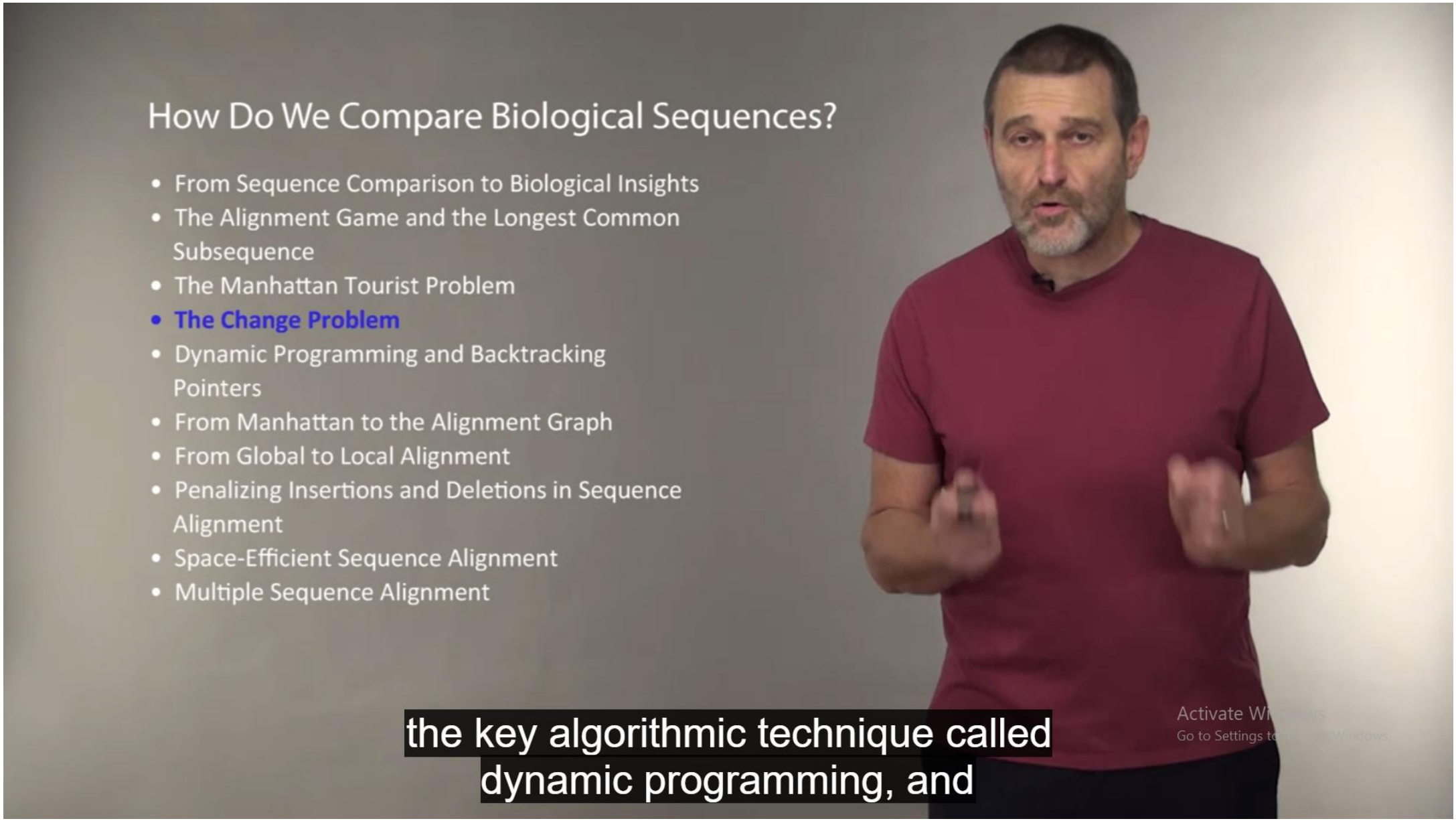
Therefore, alignments are nothing but passes in the grid.

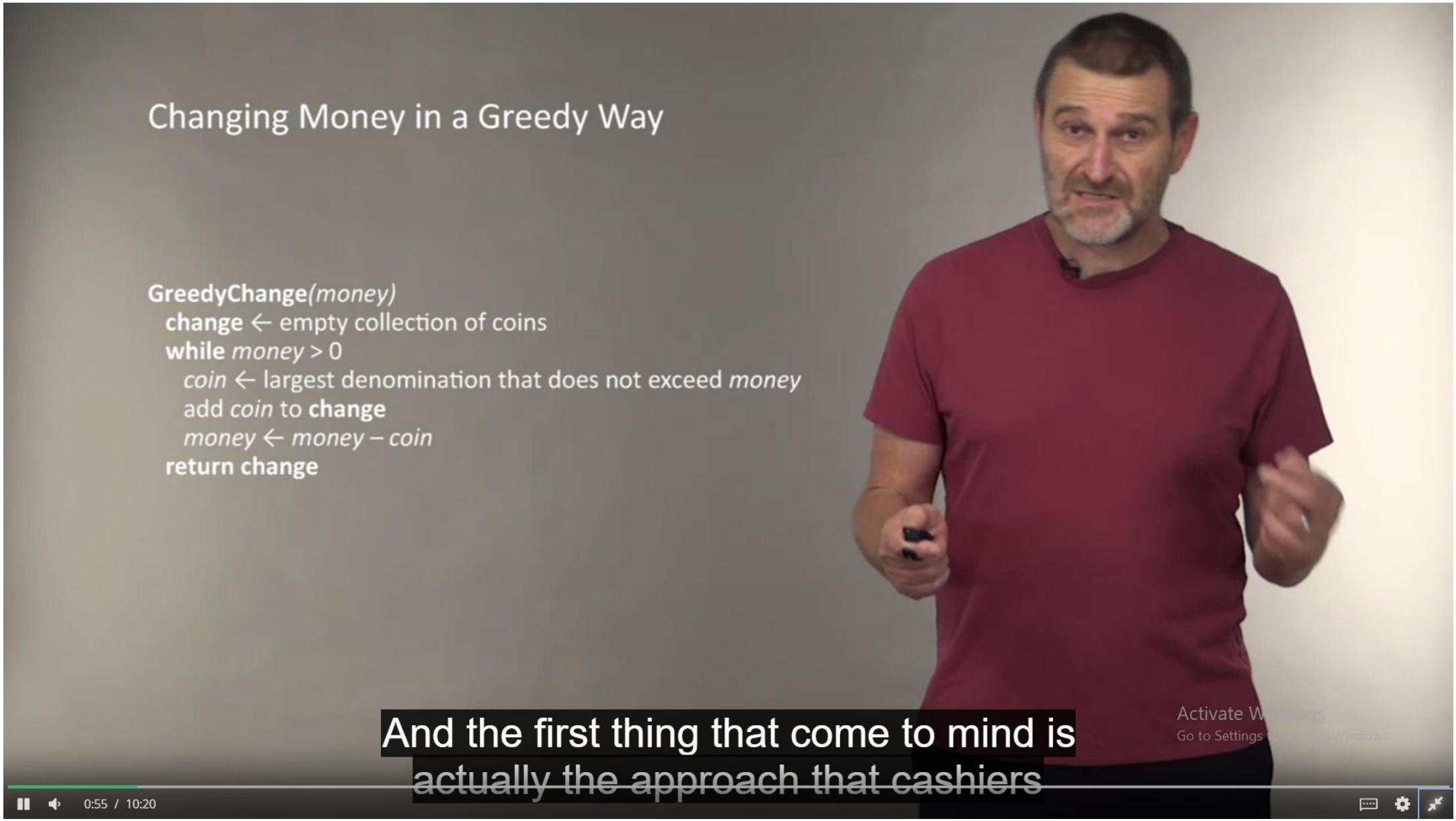
How to build a “Manhattan” for the Alignment Game?

Diagonal red edges correspond to matching symbols and have score 1



In the case of the longest common subsequence, what would be this Manhattan?


How Do We Compare Biological Sequences?
From Sequence Comparison to Biological Insights
The Alignment Game and the Longest Common Subsequence
The Manhattan Tourist Problem
The Change Problem
Dynamic Programming and Backtracking Pointers
From Manhattan to the Alignment Graph
From Global to Local Alignment
Penalizing Insertions and Deletions in Sequence Alignment
Space-Efficient Sequence Alignment
Multiple Sequence Alignment
the key algorithmic technique called dynamic programming, and
Activate Windows
Go to Settings to activate Windows.


Changing Money in a Greedy Way
GreedyChange(money)
change ← empty collection of coins
while money > 0
coin ← largest denomination that does not exceed money
add coin to change
money ← money – coin
return change
And the first thing that come to mind is
actually the approach that cashiers
0:55 / 10:20

Changing Money in Tanzania

40 cents = 25+10+5

Greedy

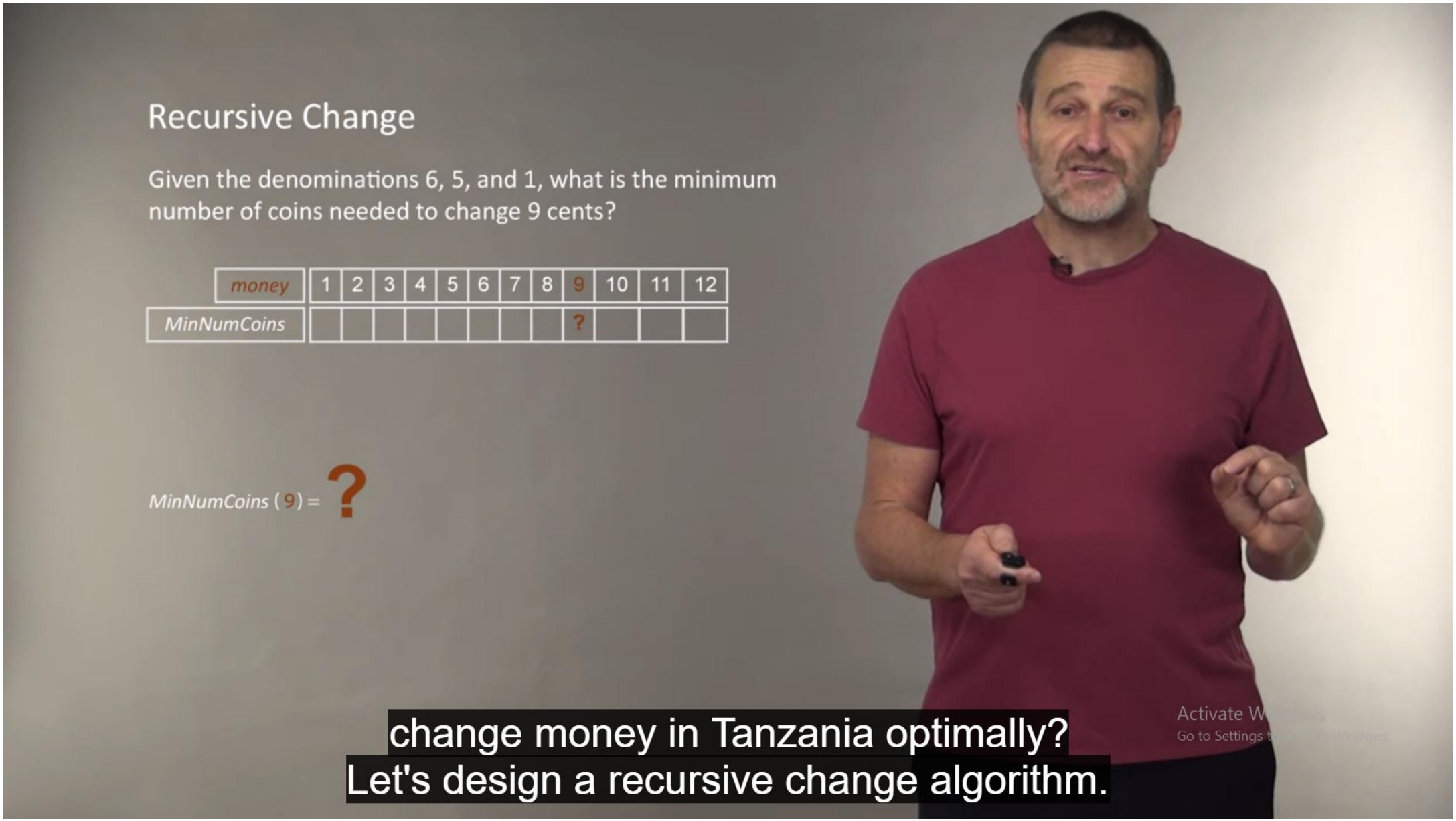
in United States, but not in Tanzania.
Tanzania has beautiful coin,

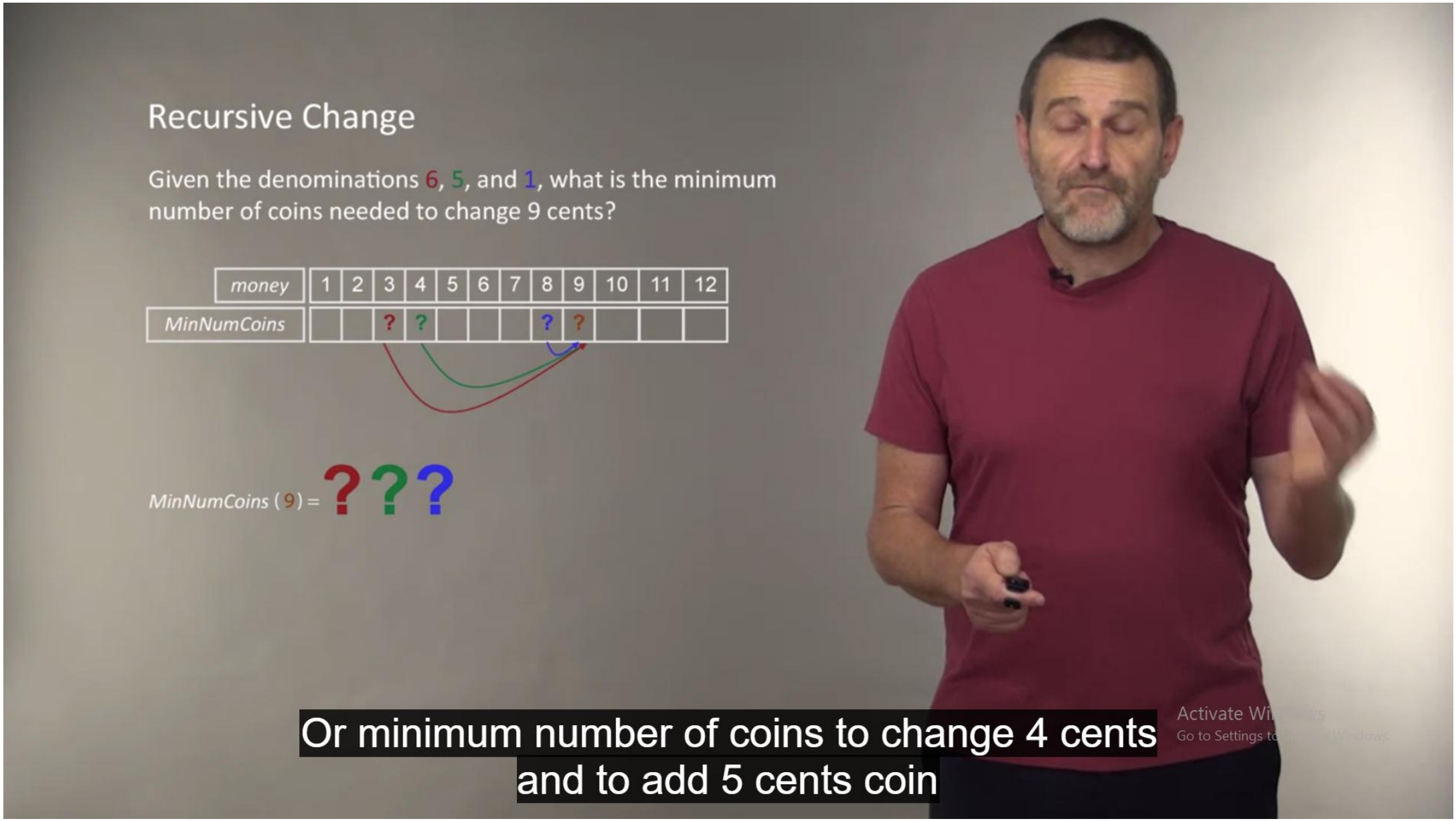
Changing Money in Tanzania:
GreedyChange Fails

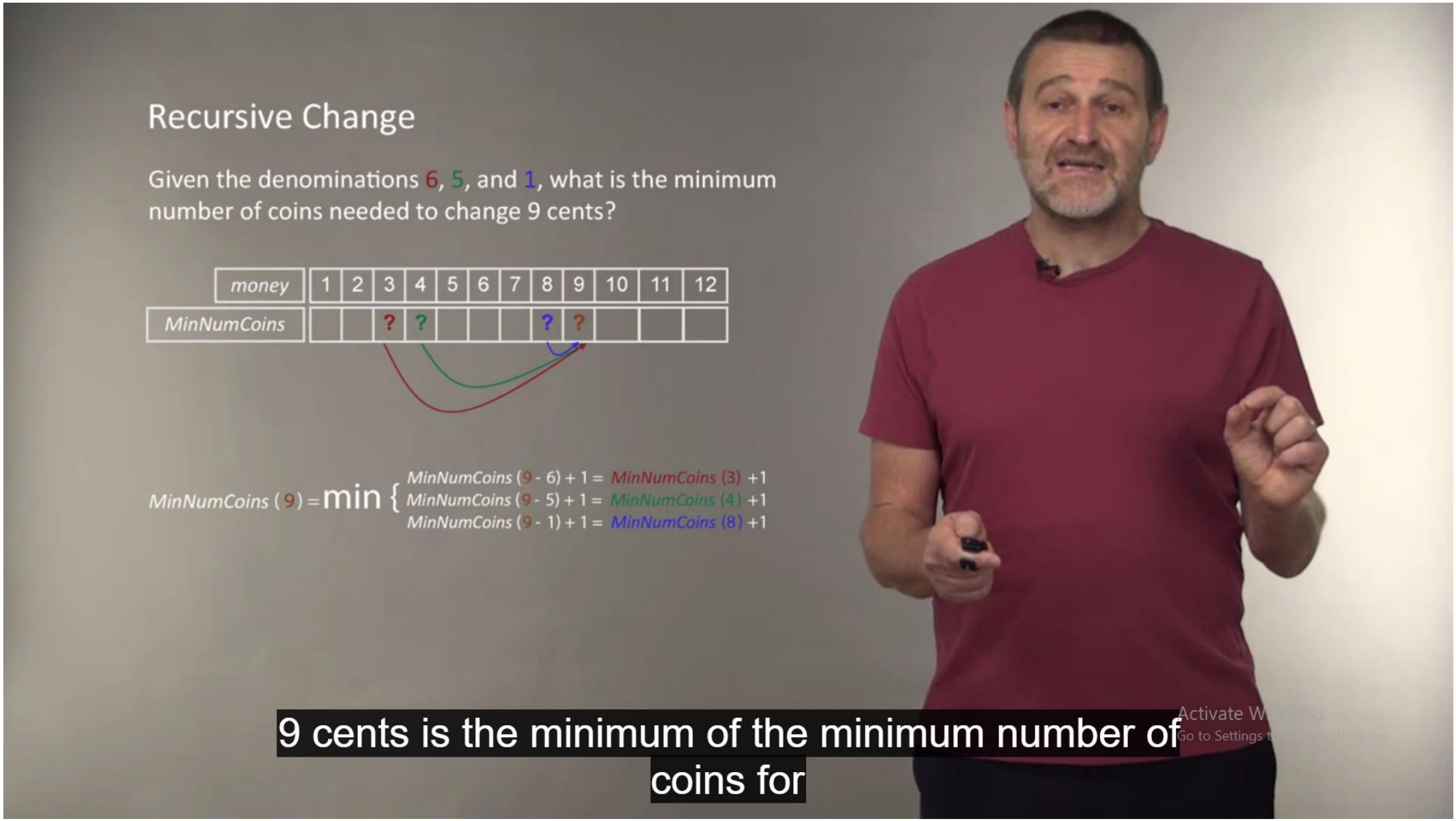
40 cents = 25+10+5 = 20+20

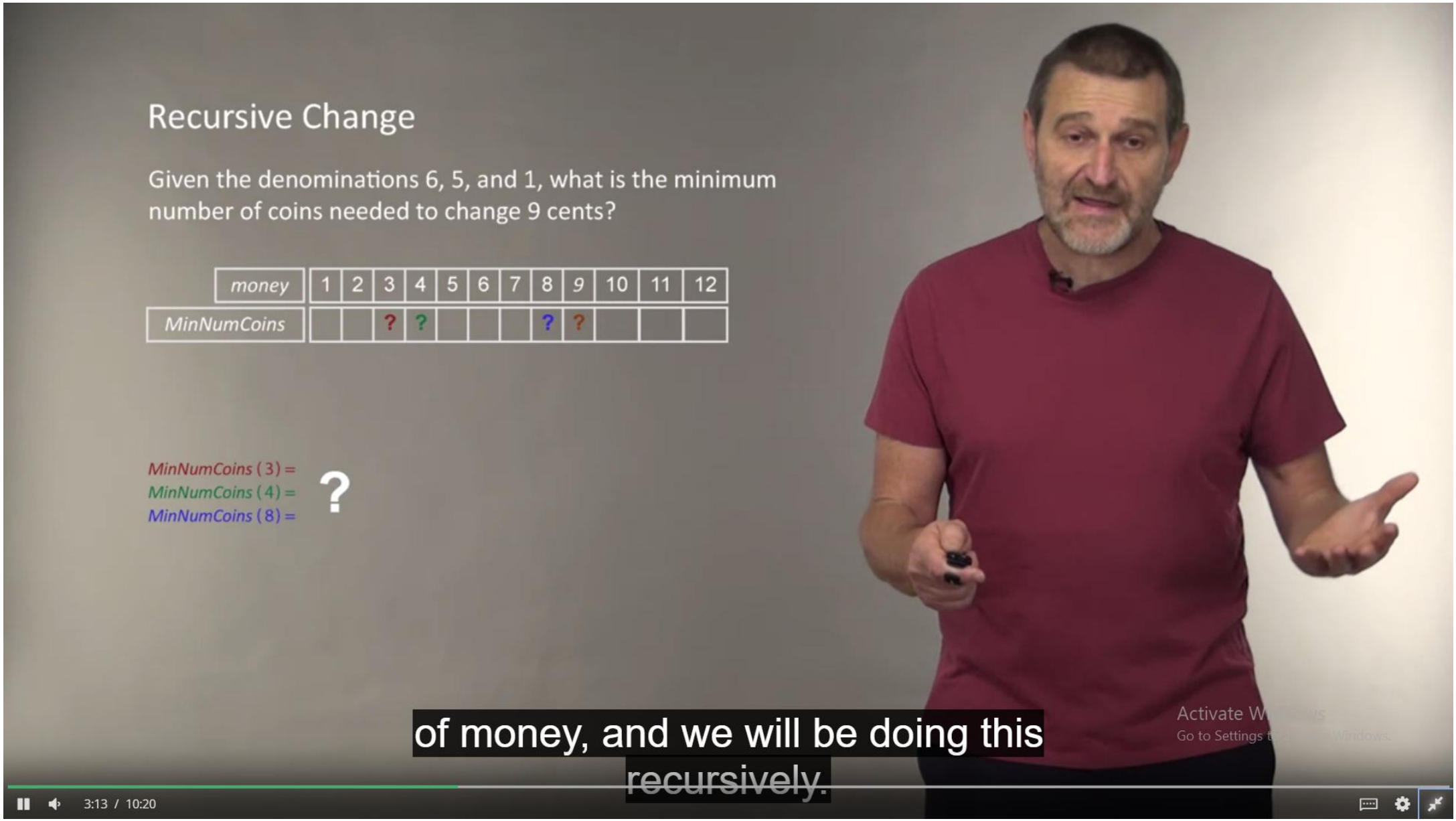
Greedy is not **Optimal**

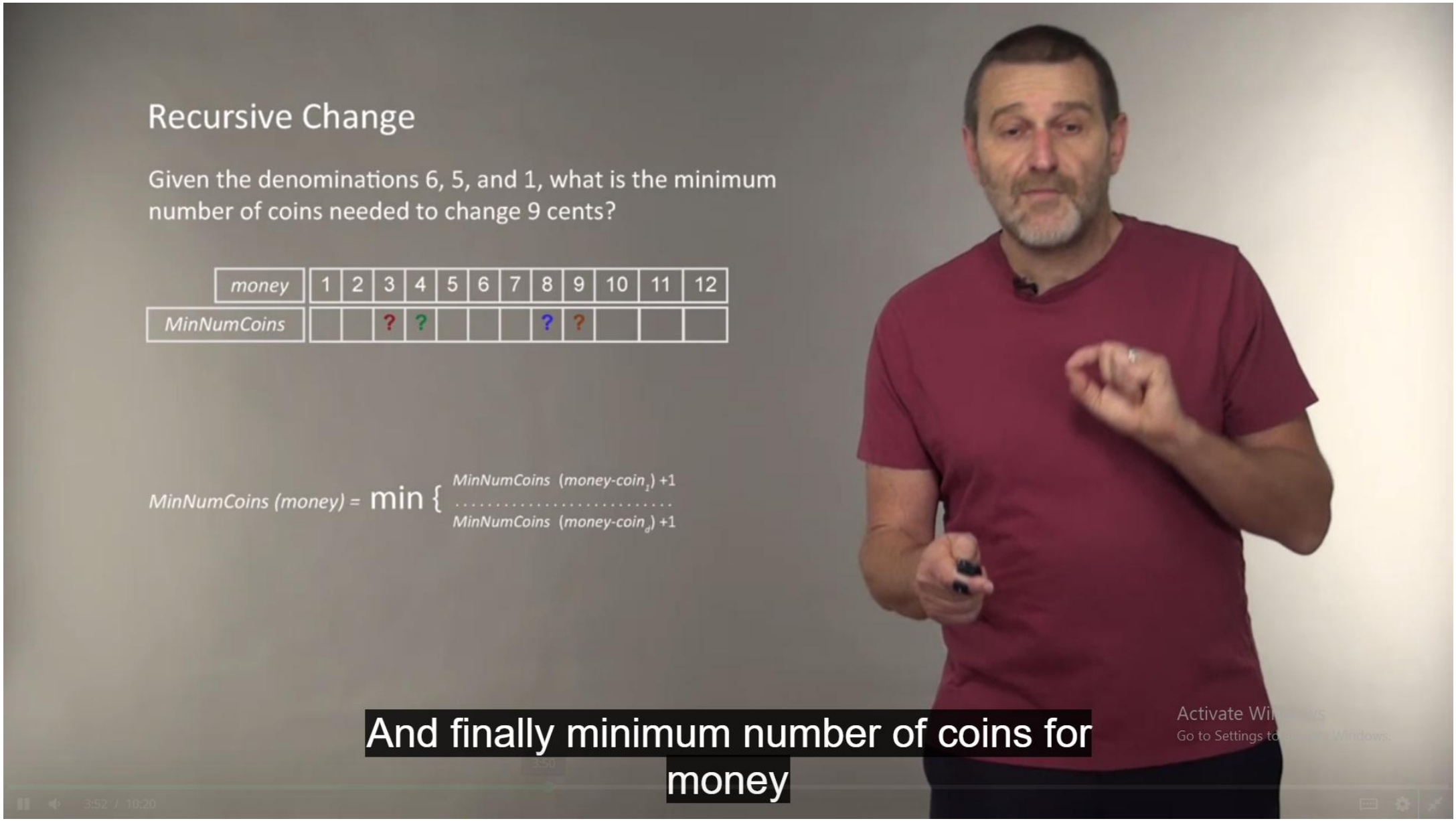
The right thing to change money in
Tanzania is simply to give you

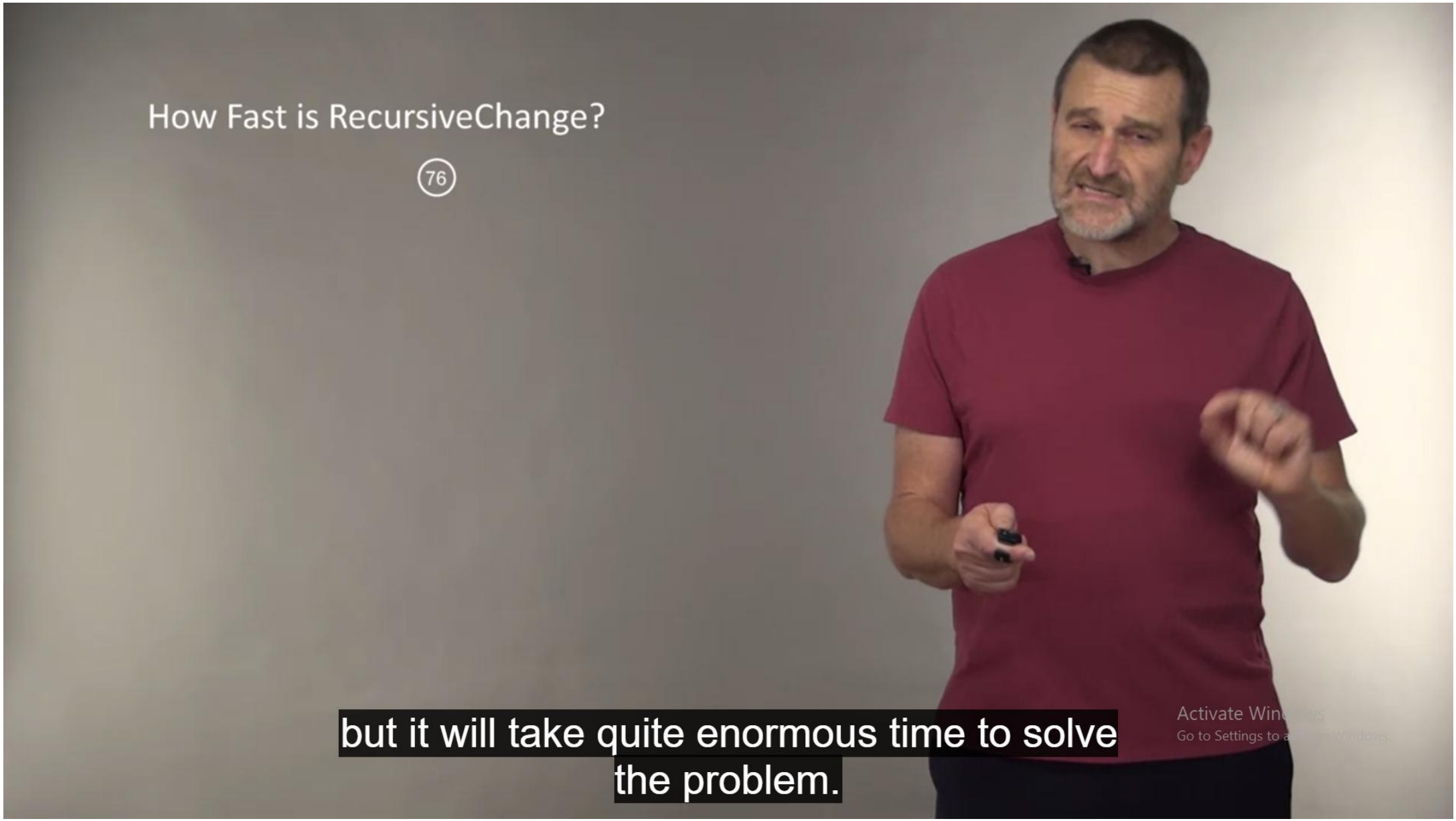

Recursive Change
Given the denominations 6, 5, and 1, what is the minimum number of coins needed to change 9 cents?
money 1 2 3 4 5 6 7 8 9 10 11 12
MinNumCoins ?
MinNumCoins (9) = ?
change money in Tanzania optimally?
Let's design a recursive change algorithm.

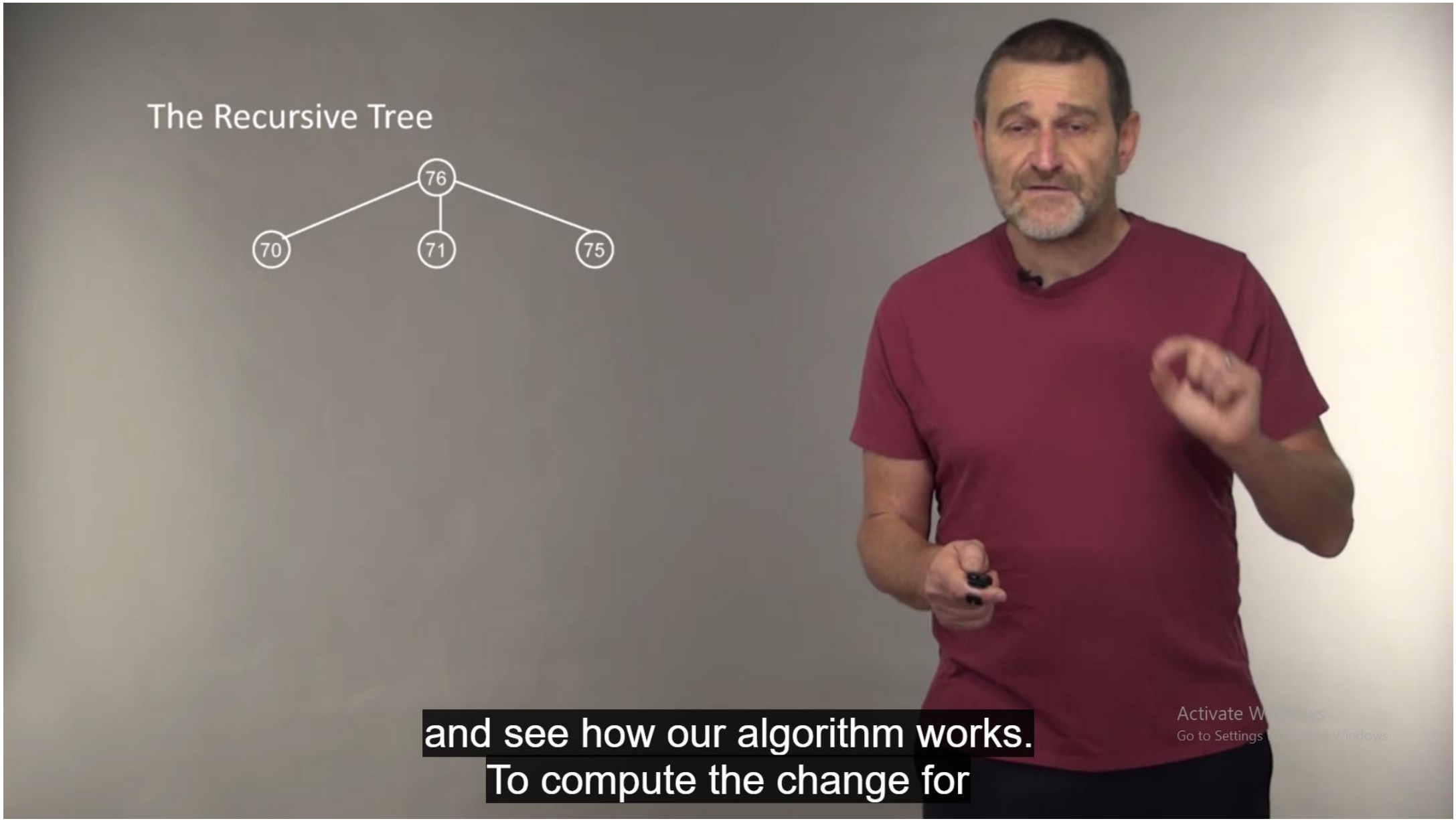

Recursive Change
Given the denominations 6, 5, and 1, what is the minimum number of coins needed to change 9 cents?
money
1
2
3
4
5
6
7
8
9
10
11
12
MinNumCoins
?
?
?
?
MinNumCoins (9) = ? ? ?
Or minimum number of coins to change 4 cents and to add 5 cents coin
Activate Windows
Go to Settings to activate Windows.

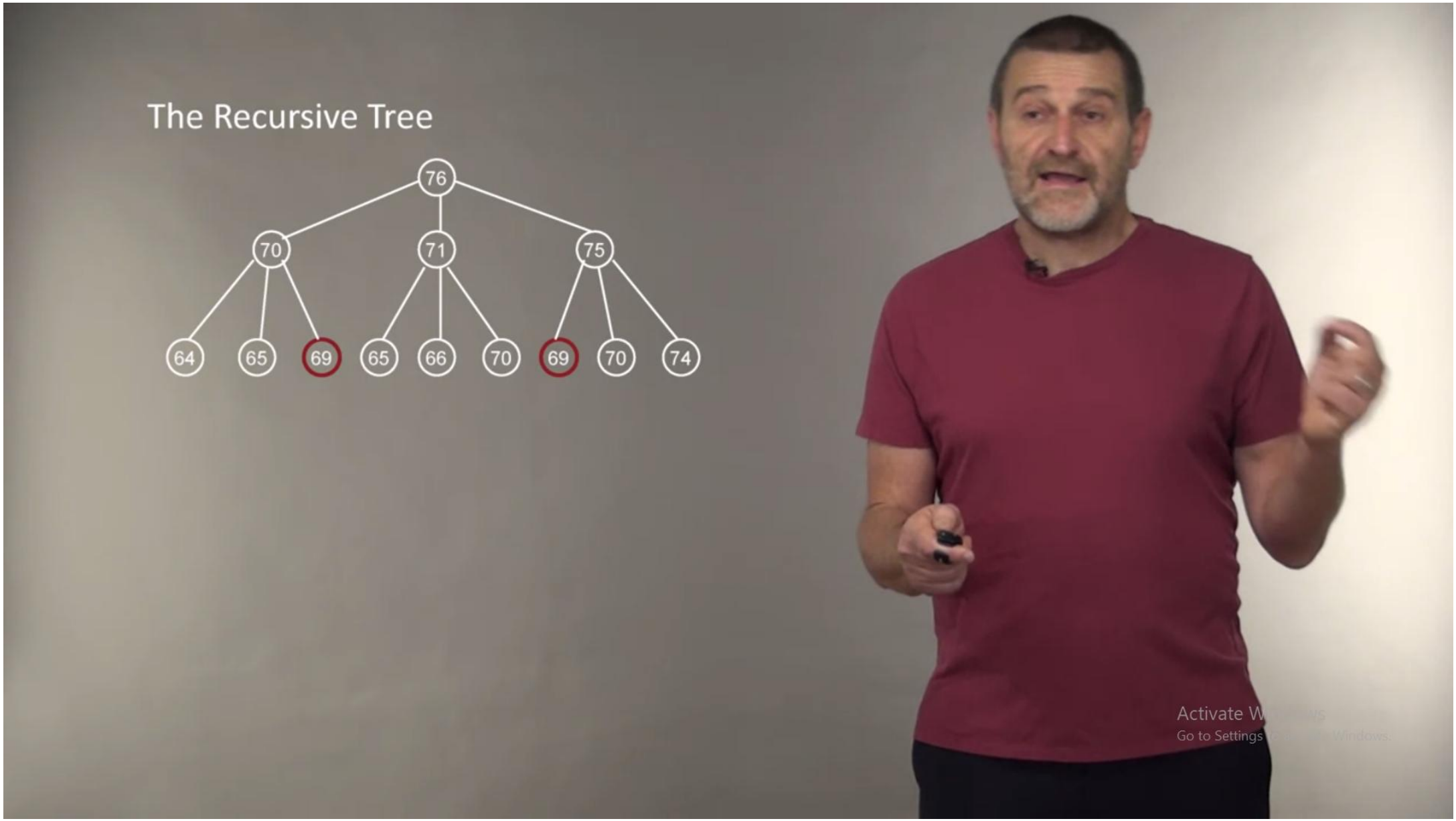

Recursive Change
Given the denominations 6, 5, and 1, what is the minimum number of coins needed to change 9 cents?
money 1 2 3 4 5 6 7 8 9 10 11 12
MinNumCoins ? ? ? ?
MinNumCoins (9) = min { MinNumCoins (9 - 6) + 1 = MinNumCoins (3) +1
MinNumCoins (9 - 5) + 1 = MinNumCoins (4) +1
MinNumCoins (9 - 1) + 1 = MinNumCoins (8) +1
9 cents is the minimum of the minimum number of coins for


Recursive Change
Given the denominations 6, 5, and 1, what is the minimum number of coins needed to change 9 cents?
money 1 2 3 4 5 6 7 8 9 10 11 12
MinNumCoins ? ? ? ?
MinNumCoins (3) =
MinNumCoins (4) =
MinNumCoins (8) =
?
of money, and we will be doing this recursively.
3:13 / 10:20


Recursive Change
Given the denominations 6, 5, and 1, what is the minimum number of coins needed to change 9 cents?
money 1 2 3 4 5 6 7 8 9 10 11 12
MinNumCoins ? ? ? ?
MinNumCoins (money) = min { MinNumCoins (money-coin1) +1 ..... MinNumCoins (money-coind) +1
And finally minimum number of coins for money
Activate Windows
Go to Settings to
Windows.
3:52 / 10:20

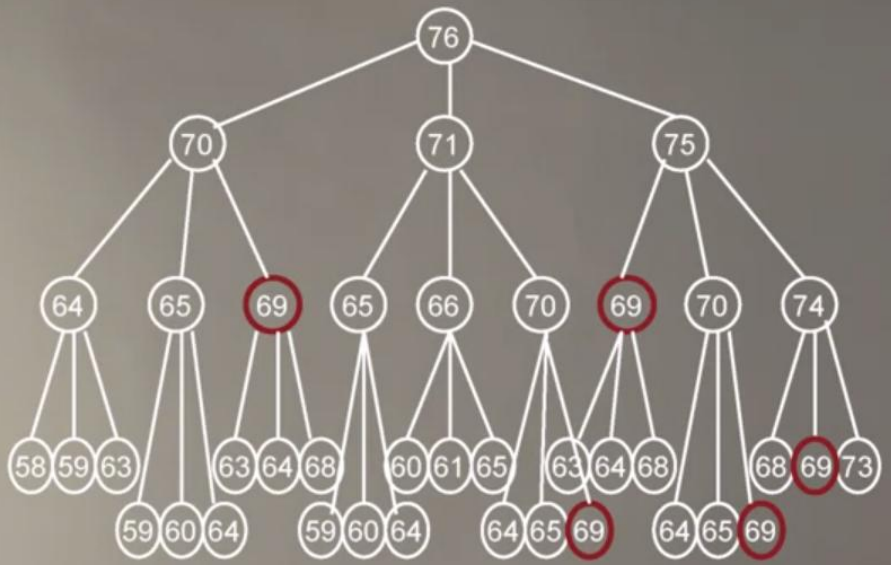

How Fast is RecursiveChange?
76
but it will take quite enormous time to solve the problem.
Activate Windows
Go to Settings to activate Windows.


The Recursive Tree
76
70
71
75
and see how our algorithm works.
To compute the change for
Activate Windows
Go to Settings

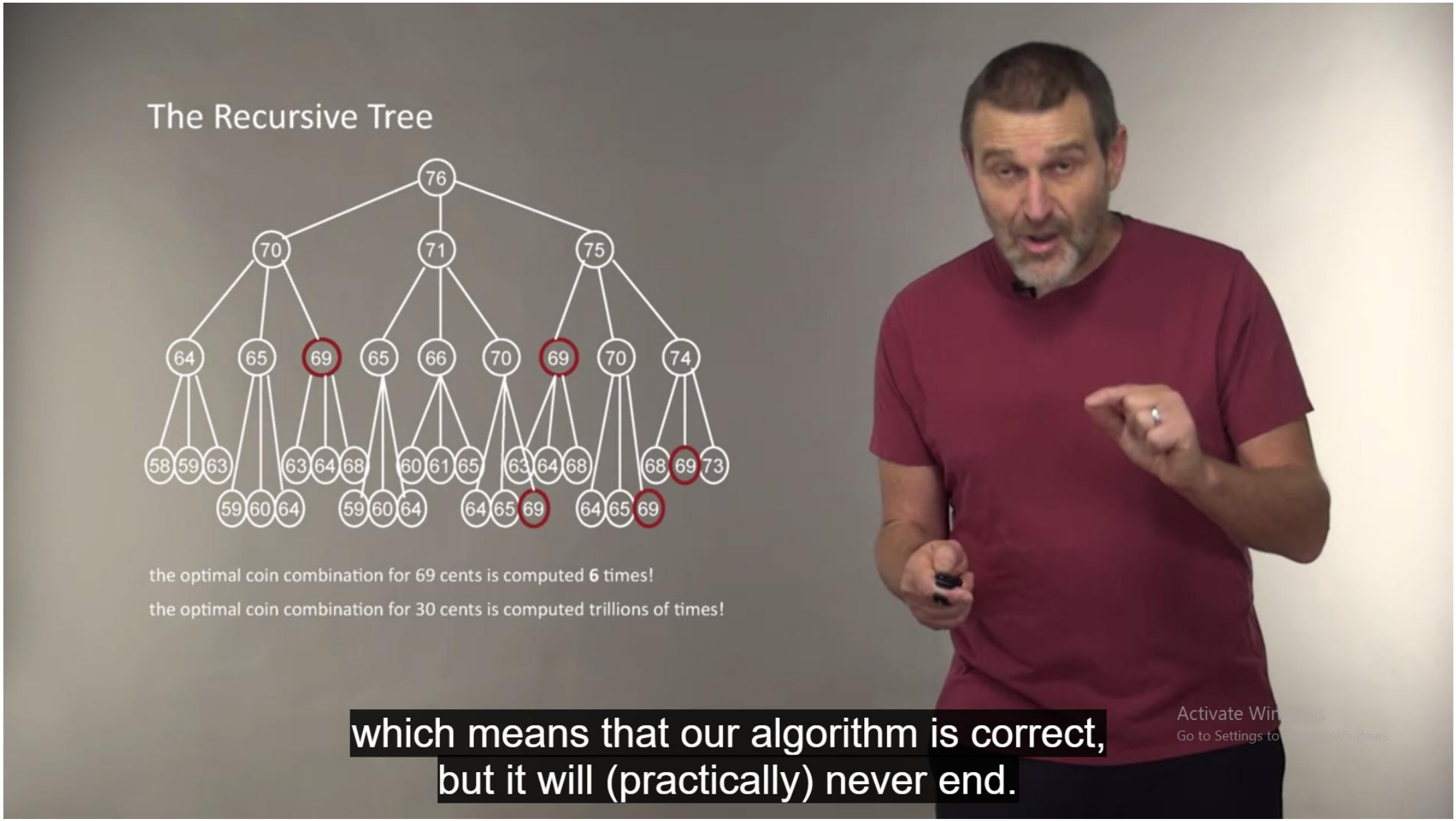

The Recursive Tree
76
70
71
75
64
65
69
65
66
70
69
70
74
Activate W
Go to Settings
Windows.


The Recursive Tree
76
70 71 75
64 65 69 65 66 70 69 70 74
58 59 63 63 64 68 60 61 65 63 64 68 68 69 73
59 60 64 59 60 64 64 65 69 64 65 69
we actually need to compute it more than twice.
Activate Windows
Go to Settings to

The Recursive Tree



the optimal coin combination for 69 cents is computed **6** times!


The Recursive Tree
76
70
71
75
64
65
69
65
66
70
69
70
74
58
59
63
63
64
68
60
61
65
63
64
68
68
69
73
59
60
64
59
60
64
64
65
69
64
65
69
the optimal coin combination for 69 cents is computed 6 times!
the optimal coin combination for 30 cents is computed trillions of times!
which means that our algorithm is correct,
but it will (practically) never end.
Activate Windows
Go to Settings to activate Windows.

Changing Money by Dynamic Programming

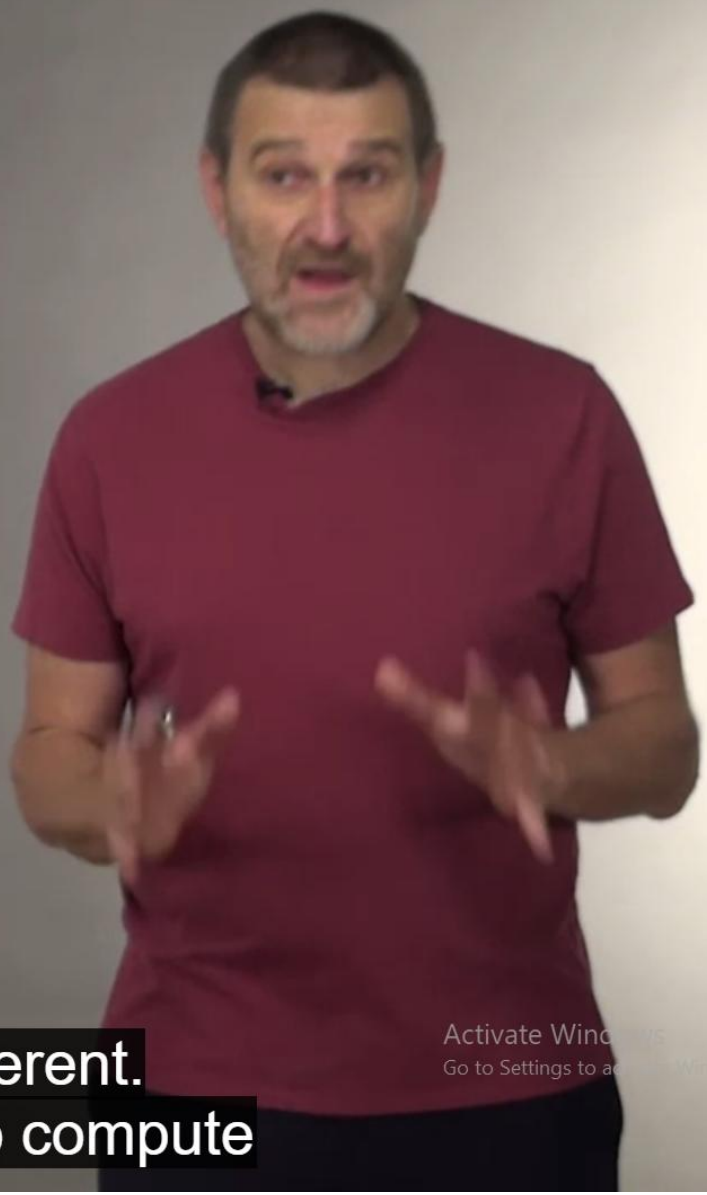
Hint. Compute all the values of

MinNumCoins(money – coin$_i$)

by the time we need to compute

MinNumCoins(money)

Richard Bellman



let's try something different.
Wouldn't it be wonderful to compute

Changing Money by Dynamic Programming

Hint. Compute all the values of

MinNumCoins(money – coin$_i$)

by the time we need to compute

MinNumCoins(money)

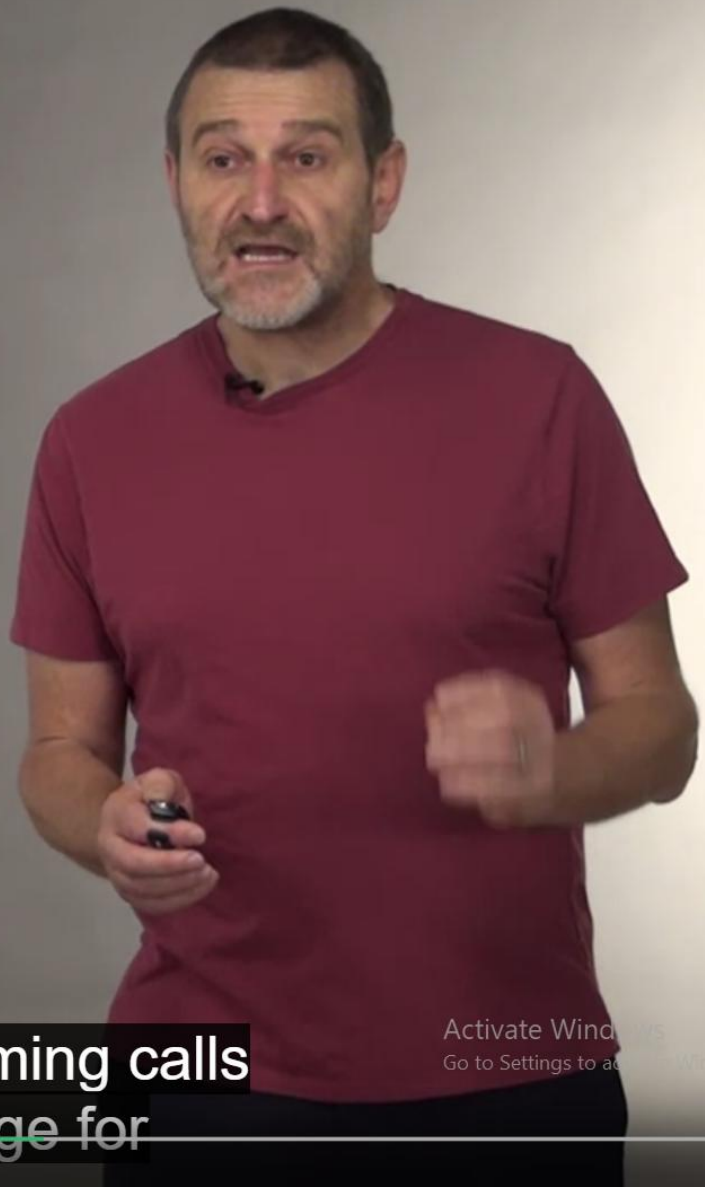
Richard Bellman

Instead of the time-consuming calls:

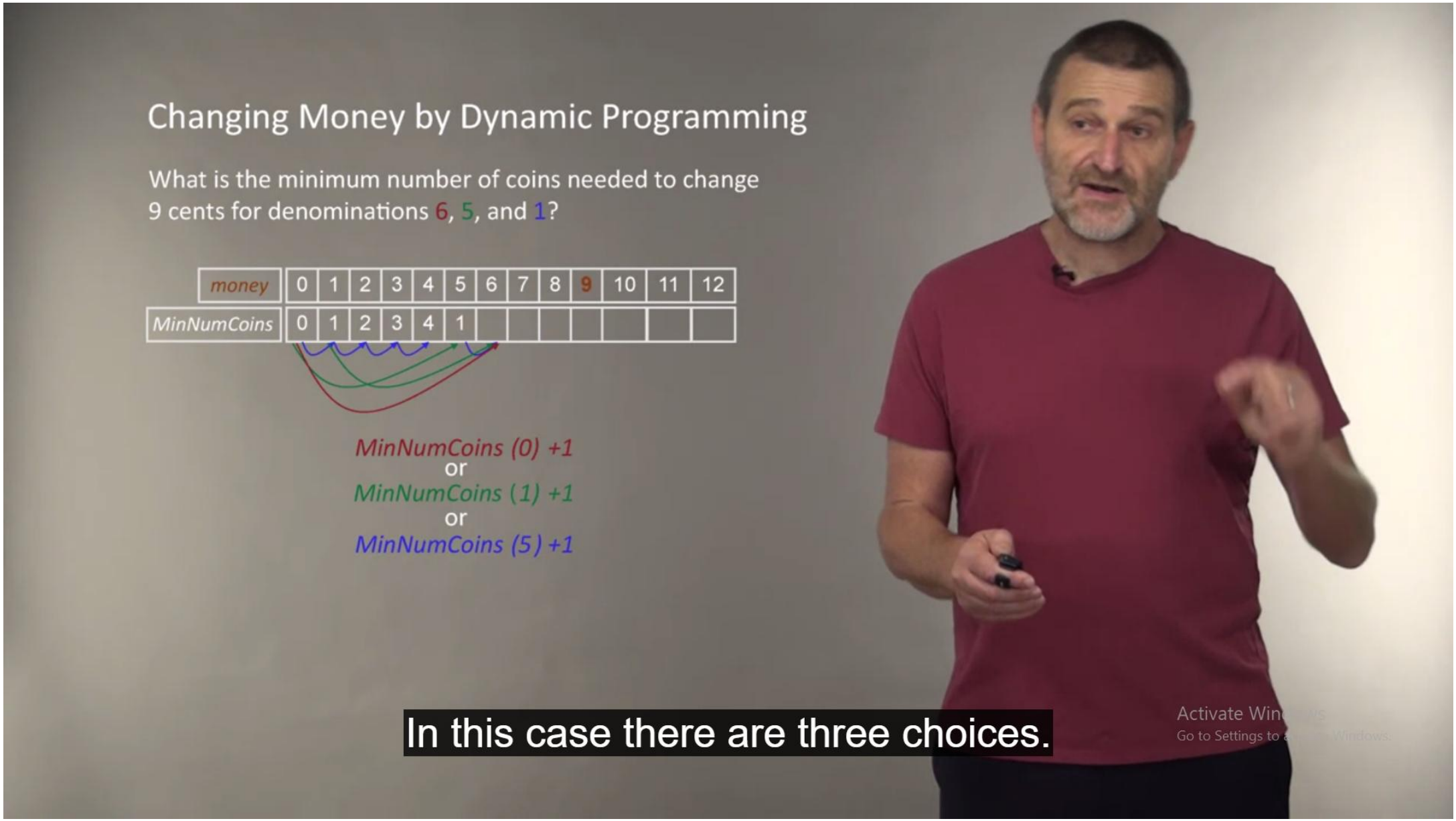
RecursiveChange(*money – coin$_i$*, ***coins***, *d)*

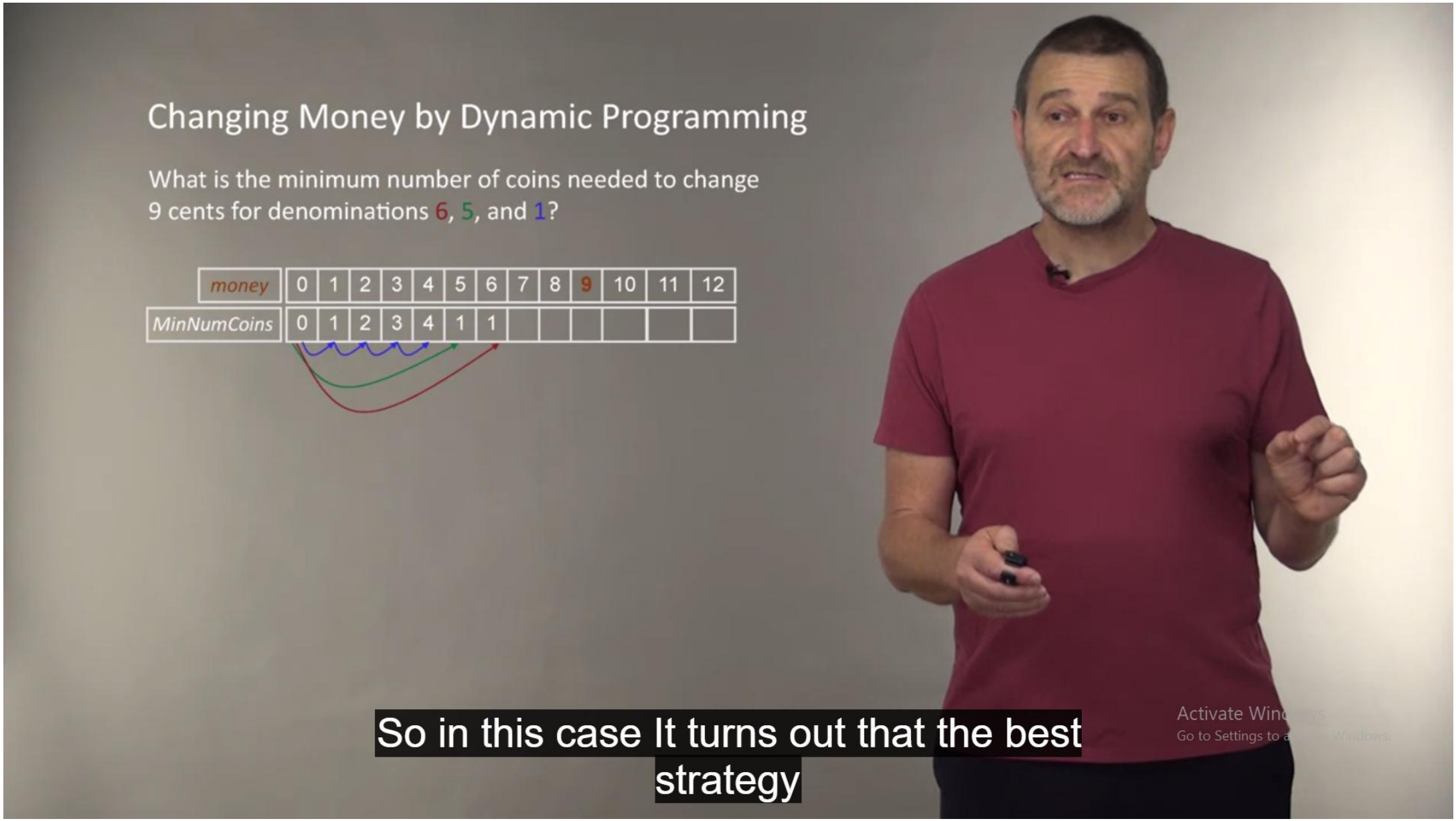
simply look up the values of

MinNumCoins(money – coin$_i$)



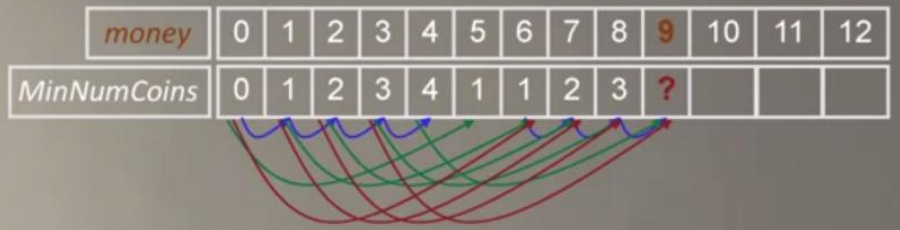
instead of the time-consuming calls
lead to recursive change for


Changing Money by Dynamic Programming
What is the minimum number of coins needed to change 9 cents for denominations 6, 5, and 1?
money 0 1 2 3 4 5 6 7 8 9 10 11 12
MinNumCoins 0 1 2 3 4 1
MinNumCoins (0) +1
or
MinNumCoins (1) +1
or
MinNumCoins (5) +1
In this case there are three choices.


Changing Money by Dynamic Programming
What is the minimum number of coins needed to change 9 cents for denominations 6, 5, and 1?
money 0 1 2 3 4 5 6 7 8 9 10 11 12
MinNumCoins 0 1 2 3 4 1 1
So in this case It turns out that the best strategy
Activate Windows
Go to Settings to activate Windows.

Changing Money by Dynamic Programming

What is the minimum number of coins needed to change 9 cents for denominations 6, 5, and 1?



“Programming” in “Dynamic Programming” Has Nothing to Do with Programming!

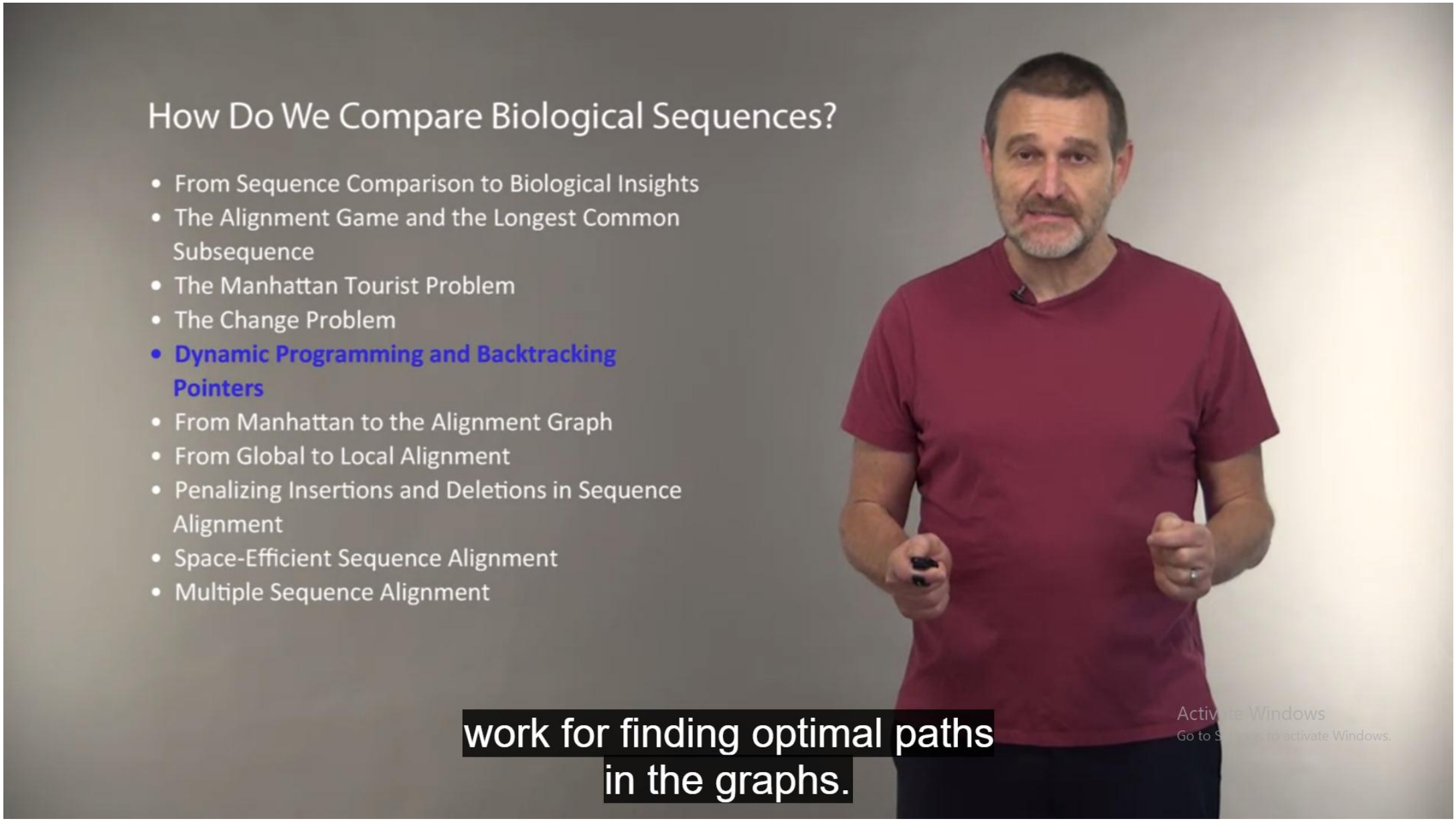
Richard Bellman developed this idea in the 1950s working on an Air Force project.

At that time, his approach seemed completely impractical.

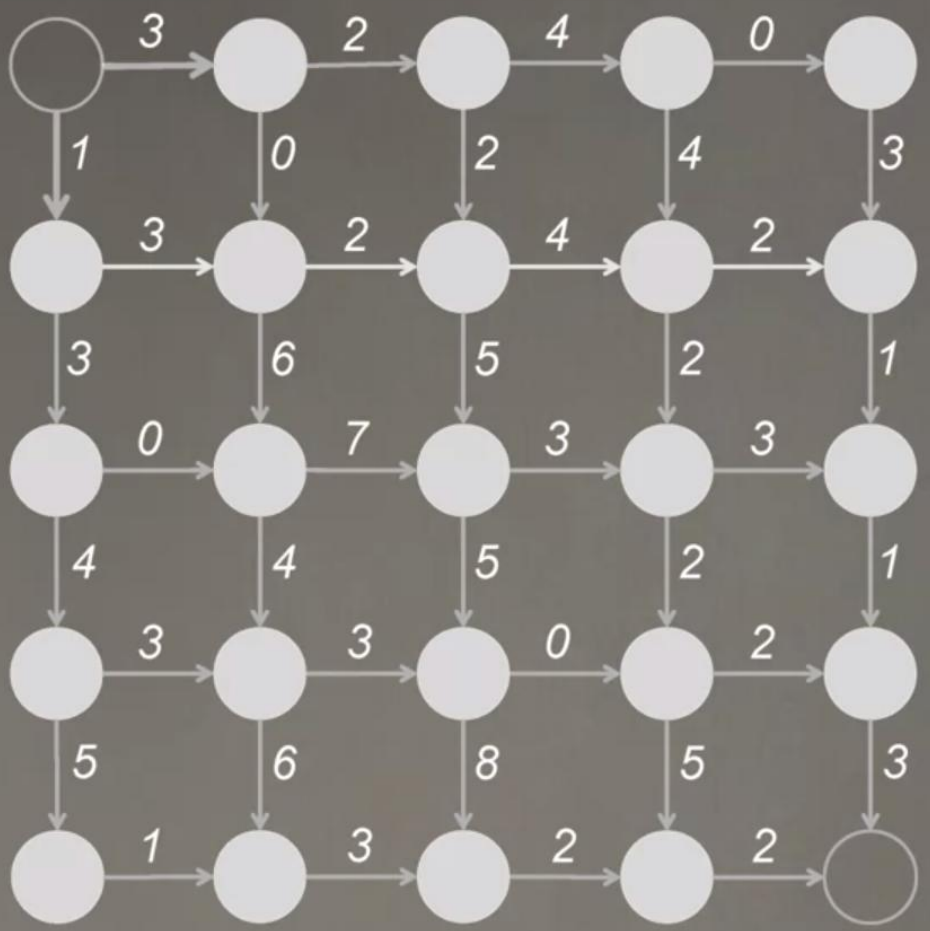
He wanted to hide that he is really doing math from the Secretary of Defense.

Richard Bellman

That's the idea developed by Richard Bellman called dynamic programming,

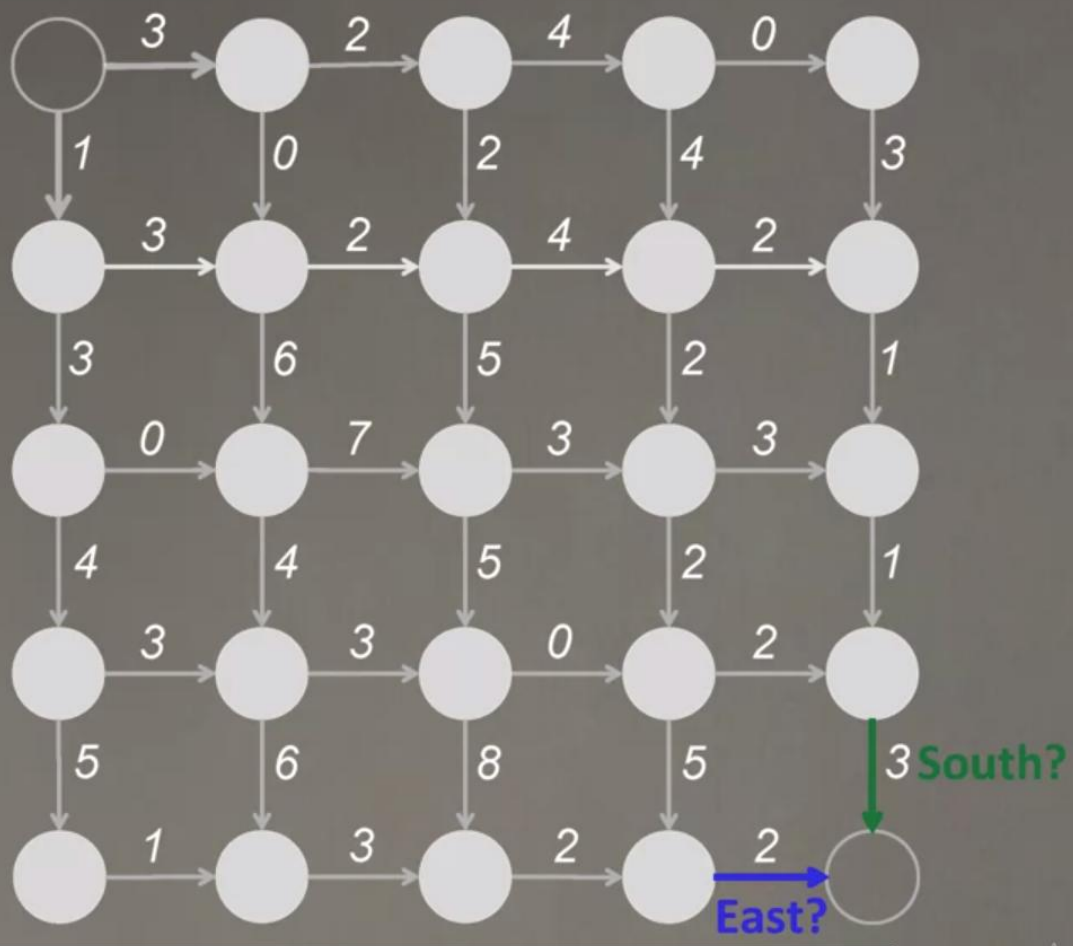

How Do We Compare Biological Sequences?
From Sequence Comparison to Biological Insights
The Alignment Game and the Longest Common Subsequence
The Manhattan Tourist Problem
The Change Problem
Dynamic Programming and Backtracking Pointers
From Manhattan to the Alignment Graph
From Global to Local Alignment
Penalizing Insertions and Deletions in Sequence Alignment
Space-Efficient Sequence Alignment
Multiple Sequence Alignment
work for finding optimal paths
in the graphs.
Activate Windows

There are only 2 ways to arrive to the sink:
by moving **South ↓**
or by moving **East →**



to reach their sink with the maximum number of attractions.

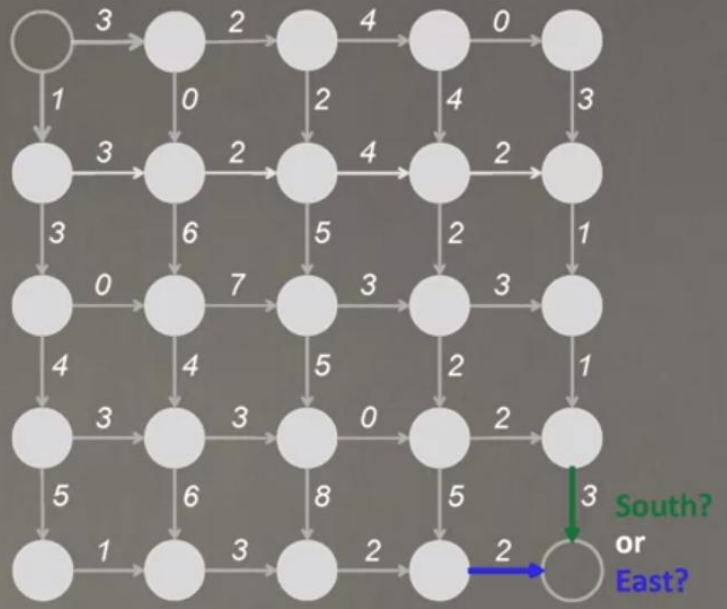
There are only 2 ways to arrive to the sink:
by moving **South ↓**
or by moving **East →**



So, to arrive to the final destination we

South or East?

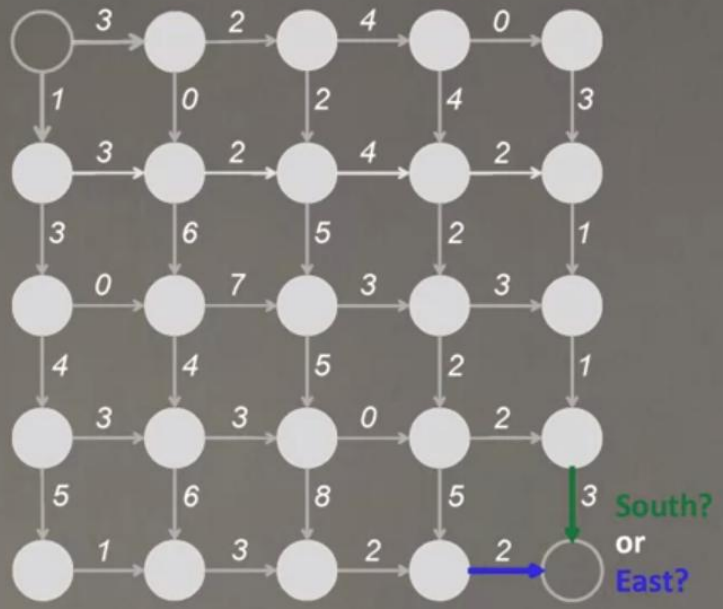
SouthOrEast(*i*,*j*):
the length of the longest path from (0,0) to (*i*,*j*)



And let's define a variable, which is

South or East?

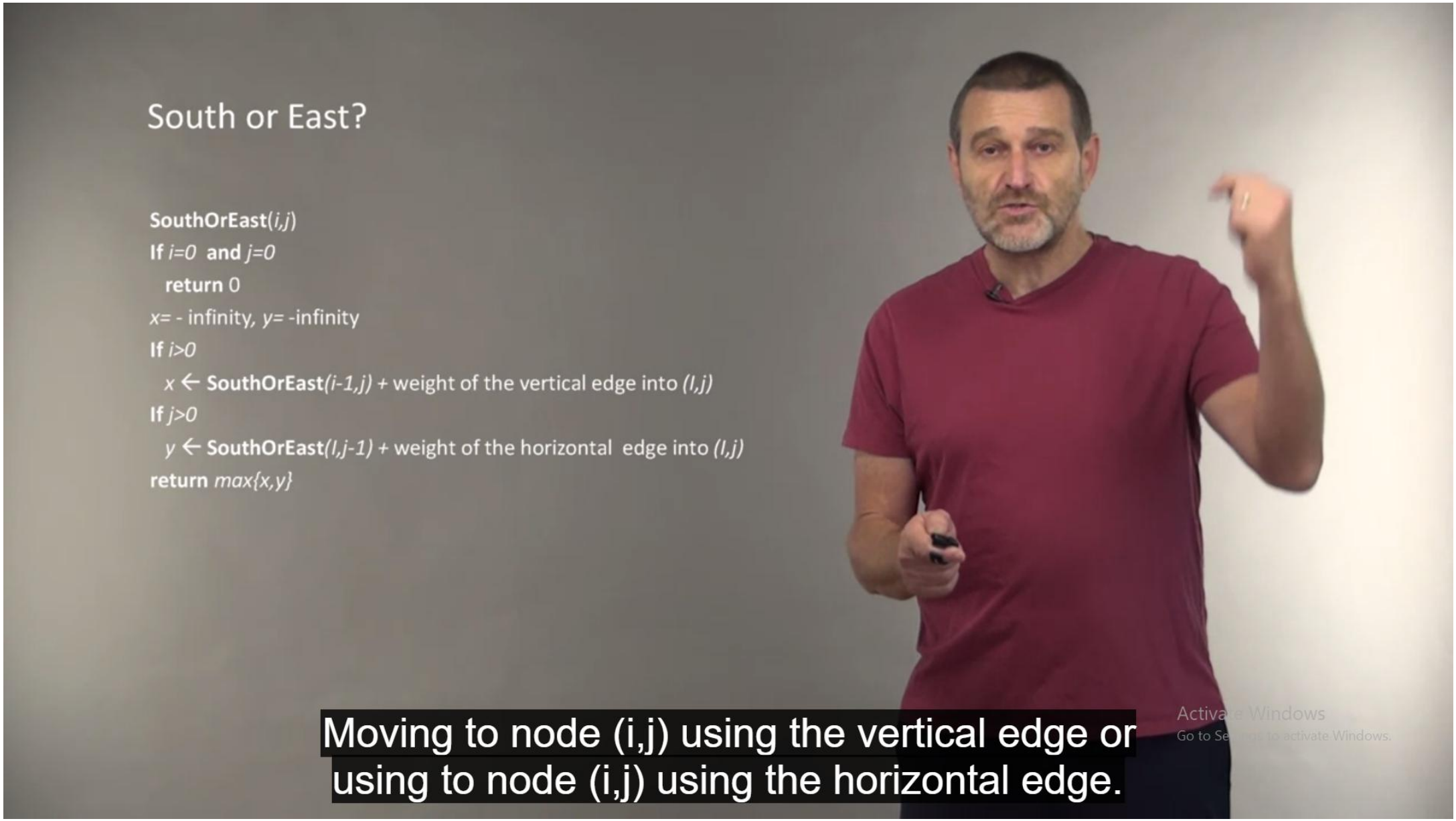
SouthOrEast(*i,j*):
the length of the longest path from (0,0) to (*i,j*)

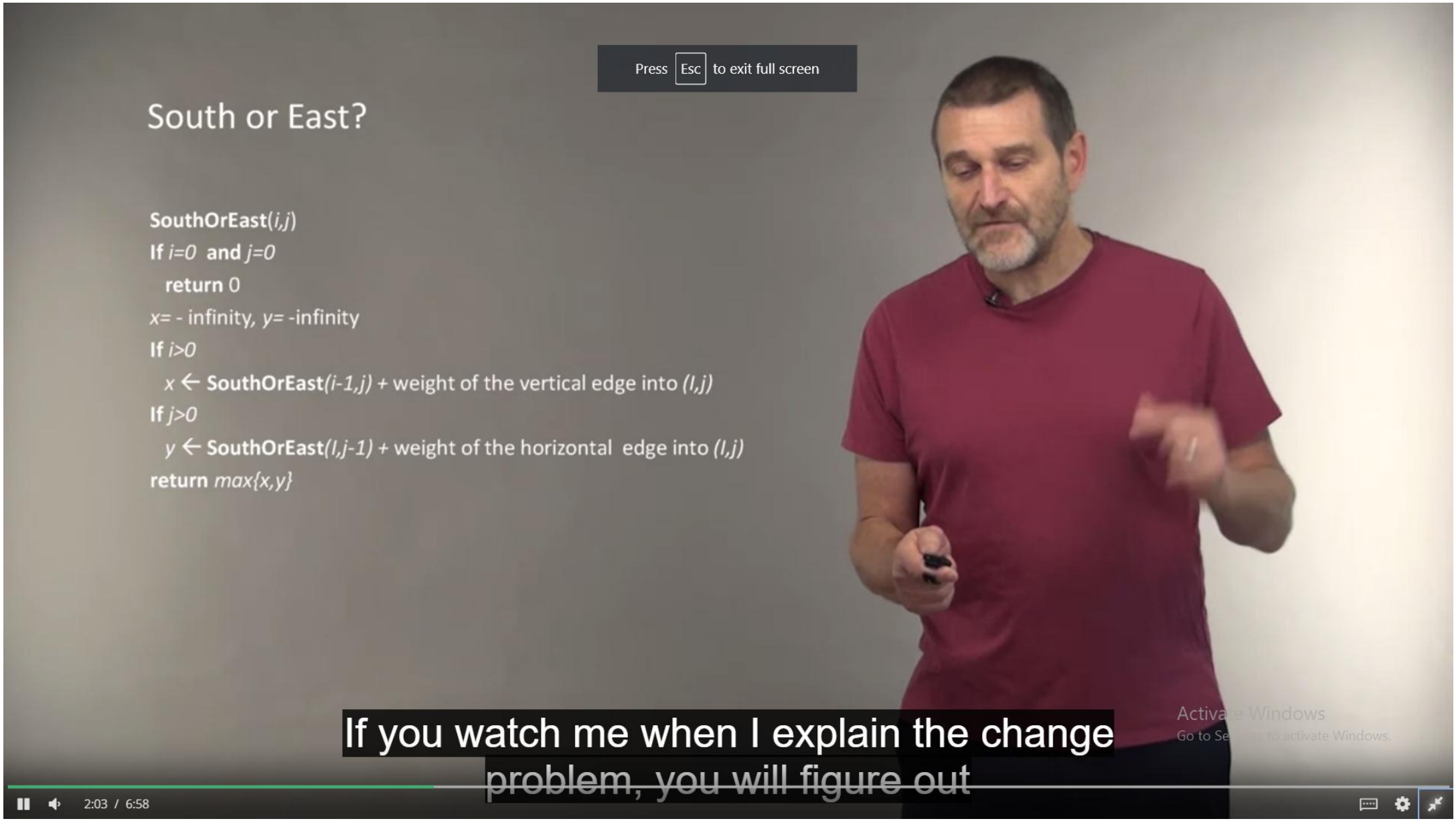


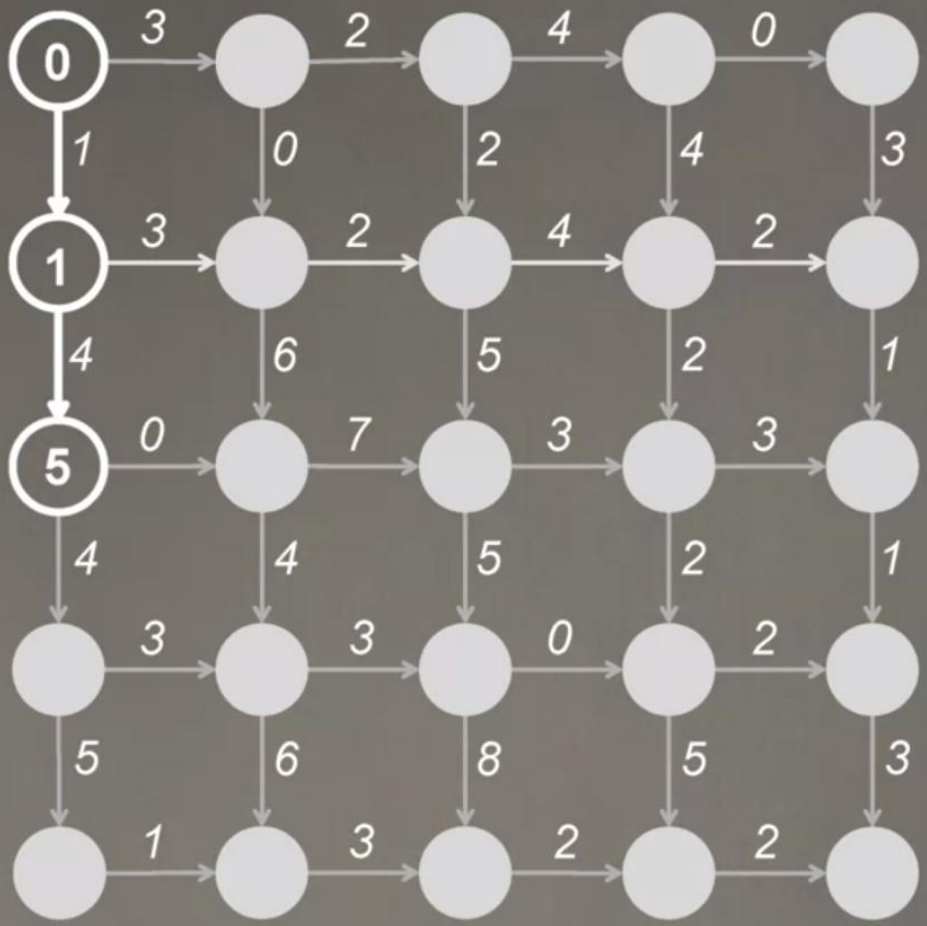
SouthOrEast(*n,m*)=

min{ **SouthOrEast**(*n-1,m*)+weight of edge "↓"into (*n,m*)
SouthOrEast(*n,m-1*)+ weight of edge "→"into (*n,m*)

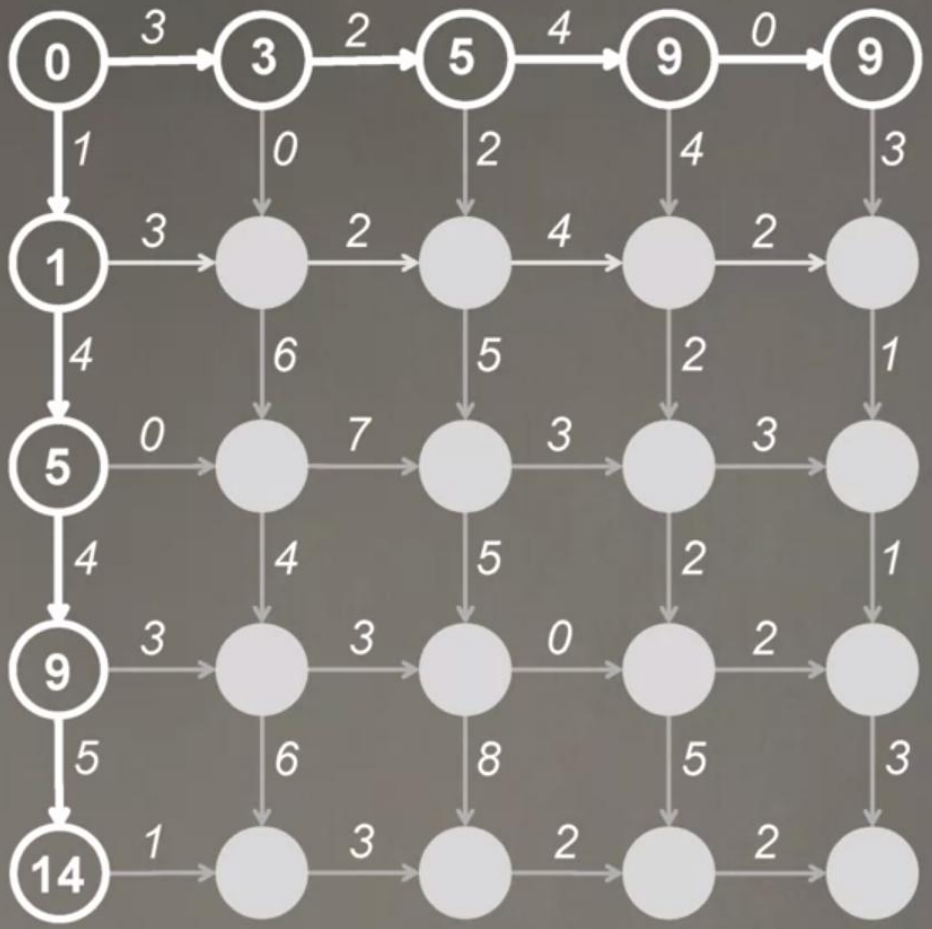
computing south or east for (n,m),
for the final destination vertex.


South or East?
SouthOrEast(i,j)
If i=0 and j=0
return 0
x= - infinity, y= -infinity
If i>0
x ← SouthOrEast(i-1,j) + weight of the vertical edge into (I,j)
If j>0
y ← SouthOrEast(I,j-1) + weight of the horizontal edge into (I,j)
return max{x,y}
Moving to node (i,j) using the vertical edge or using to node (i,j) using the horizontal edge.
Activate Windows
Go to Settings to activate Windows.


Press Esc to exit full screen
South or East?
SouthOrEast(i,j)
If i=0 and j=0
return 0
x= - infinity, y= -infinity
If i>0
x ← SouthOrEast(i-1,j) + weight of the vertical edge into (I,j)
If j>0
y ← SouthOrEast(I,j-1) + weight of the horizontal edge into (I,j)
return max{x,y}
If you watch me when I explain the change problem, you will figure out
2:03 / 6:58

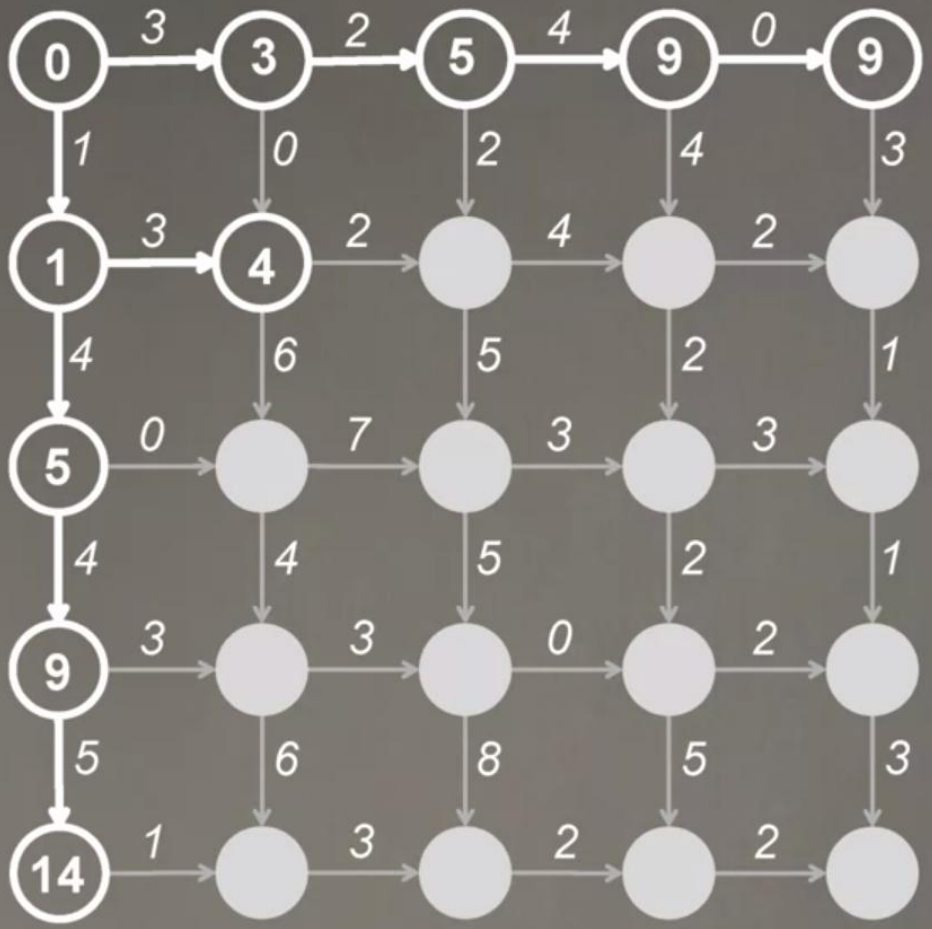


The optimal score to reach this node will be 5,



And as a result, we can compute all values we described the lengths of the longest path.

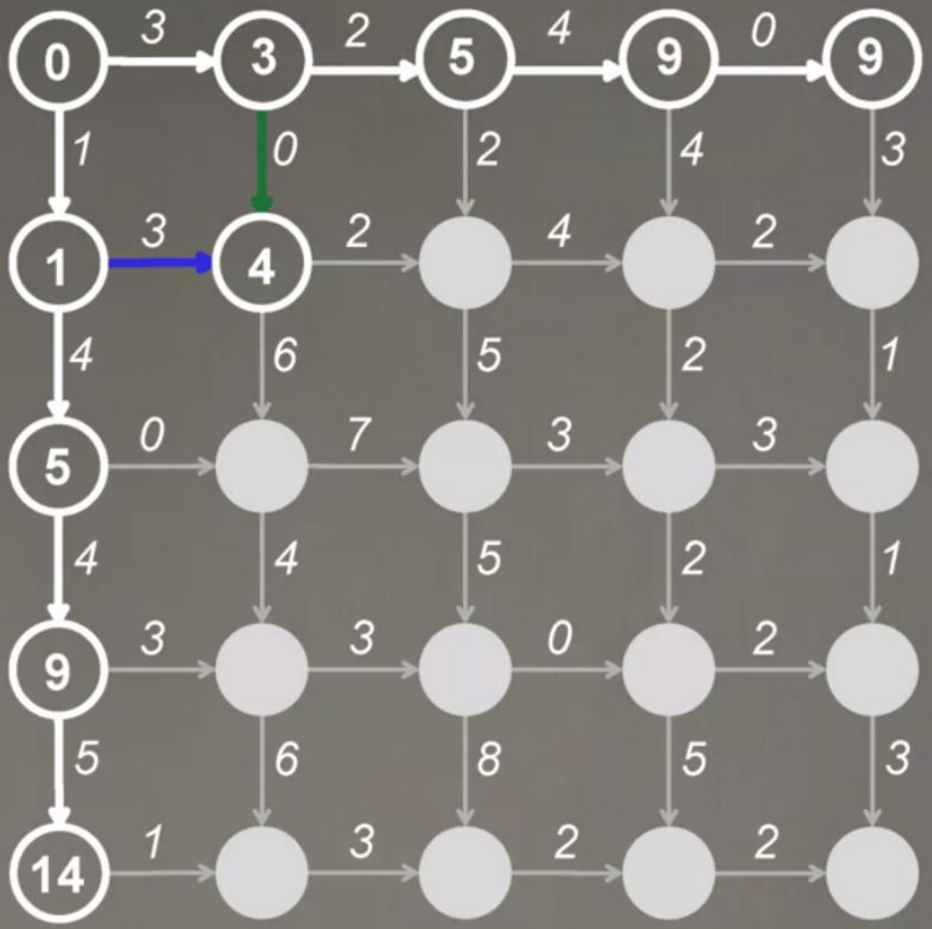
We arrived to (1,1) by the **bold** edge:



And for the future, we represent the edge

South
or
East?

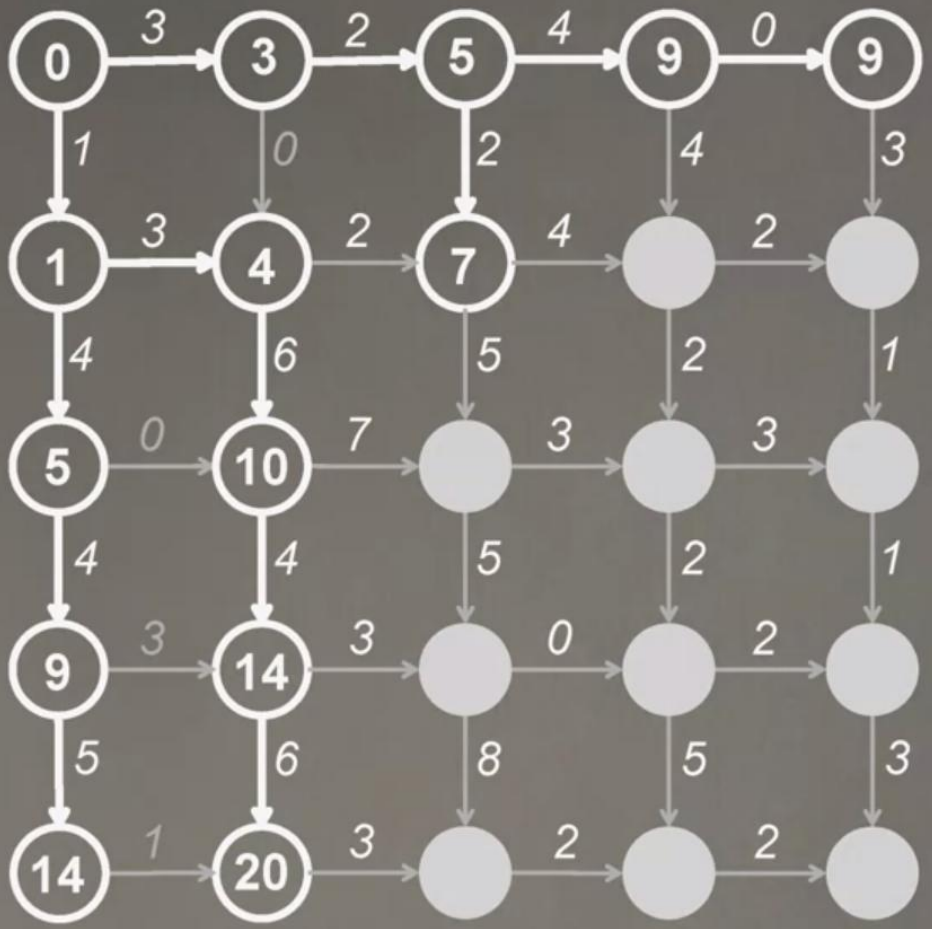
1+3 > 3+0



And we know afterwards that
the optimal path to this node has length 4.

South
or
East?

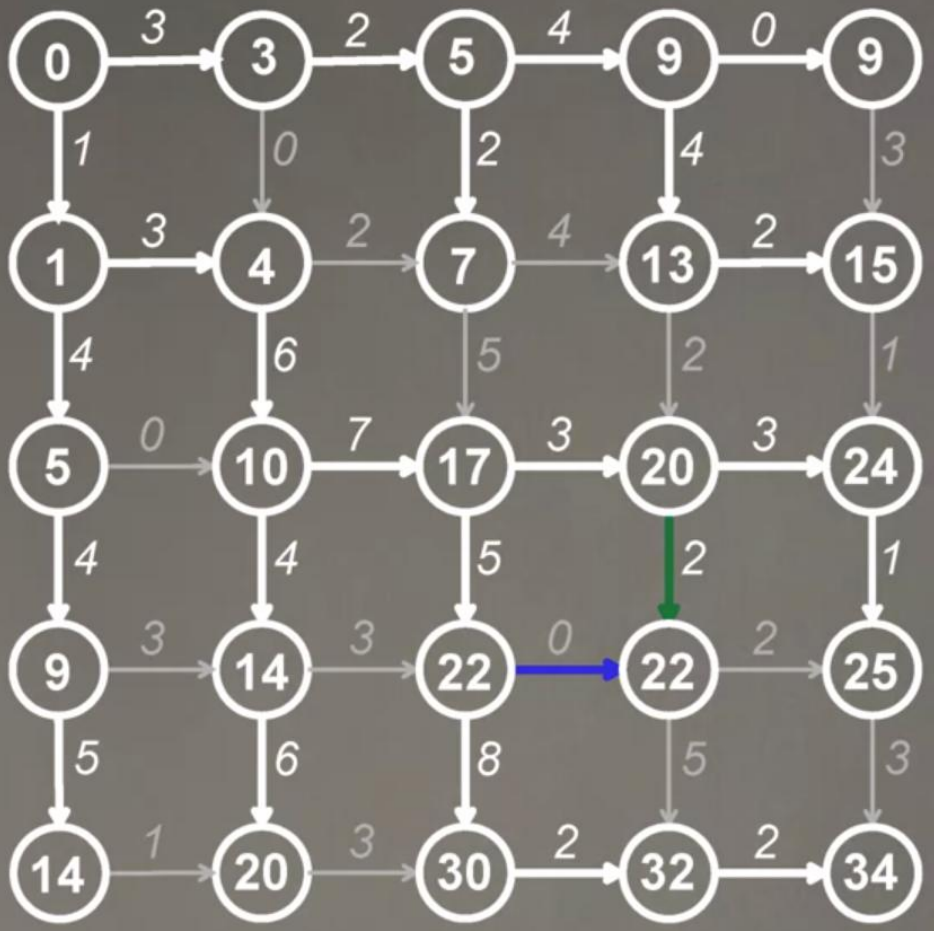
5+2 < 4+2



by moving along vertical edge.

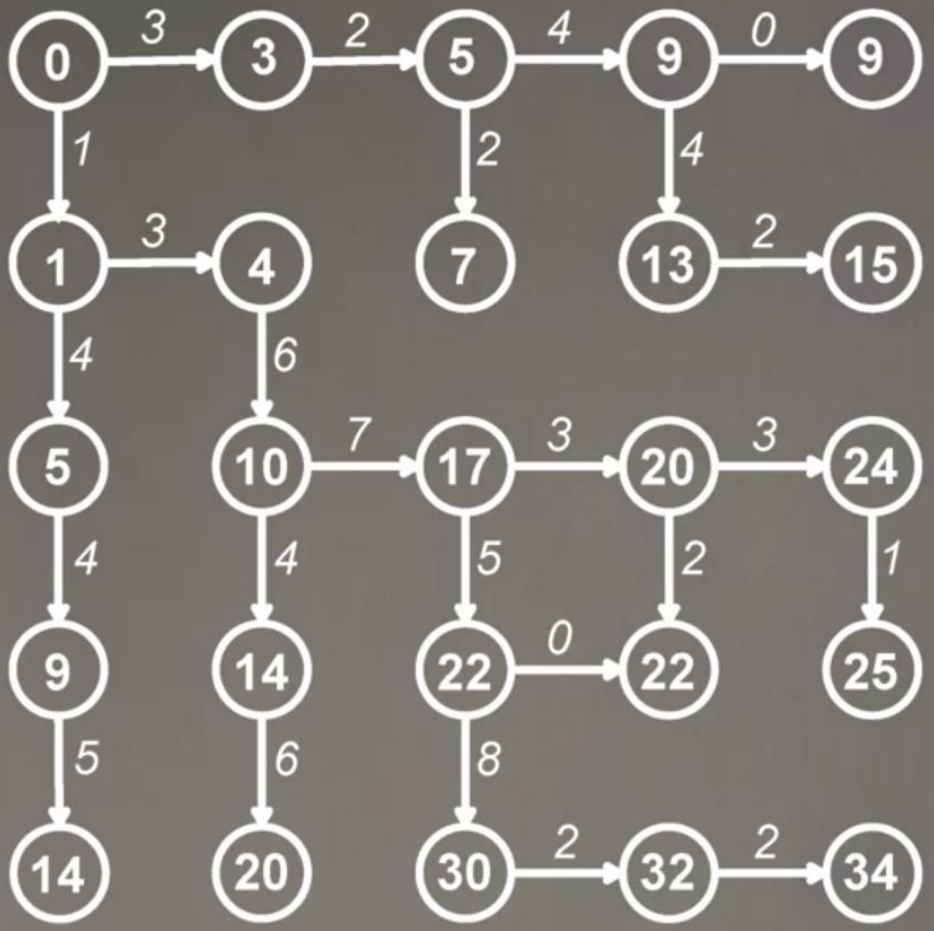
South
or
East?

22+0=20+2



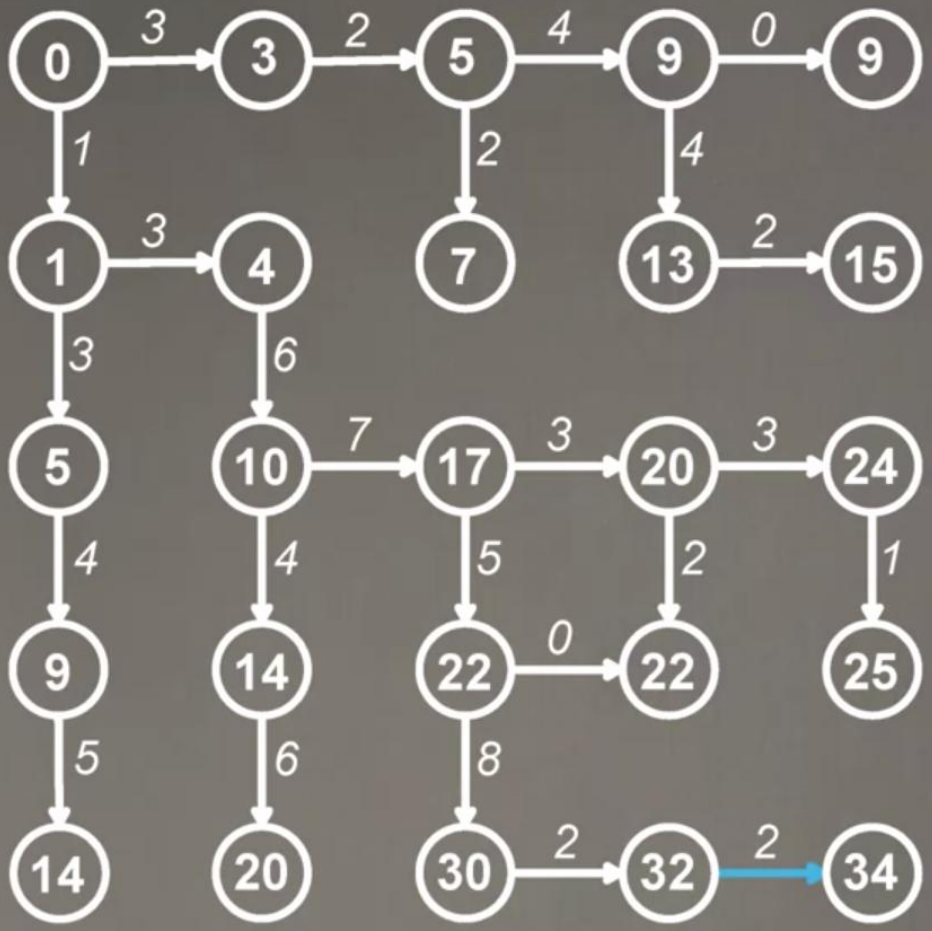
When we are doing this, please note that
in some cases, instead

Backtracking pointers: the best way to get to each node

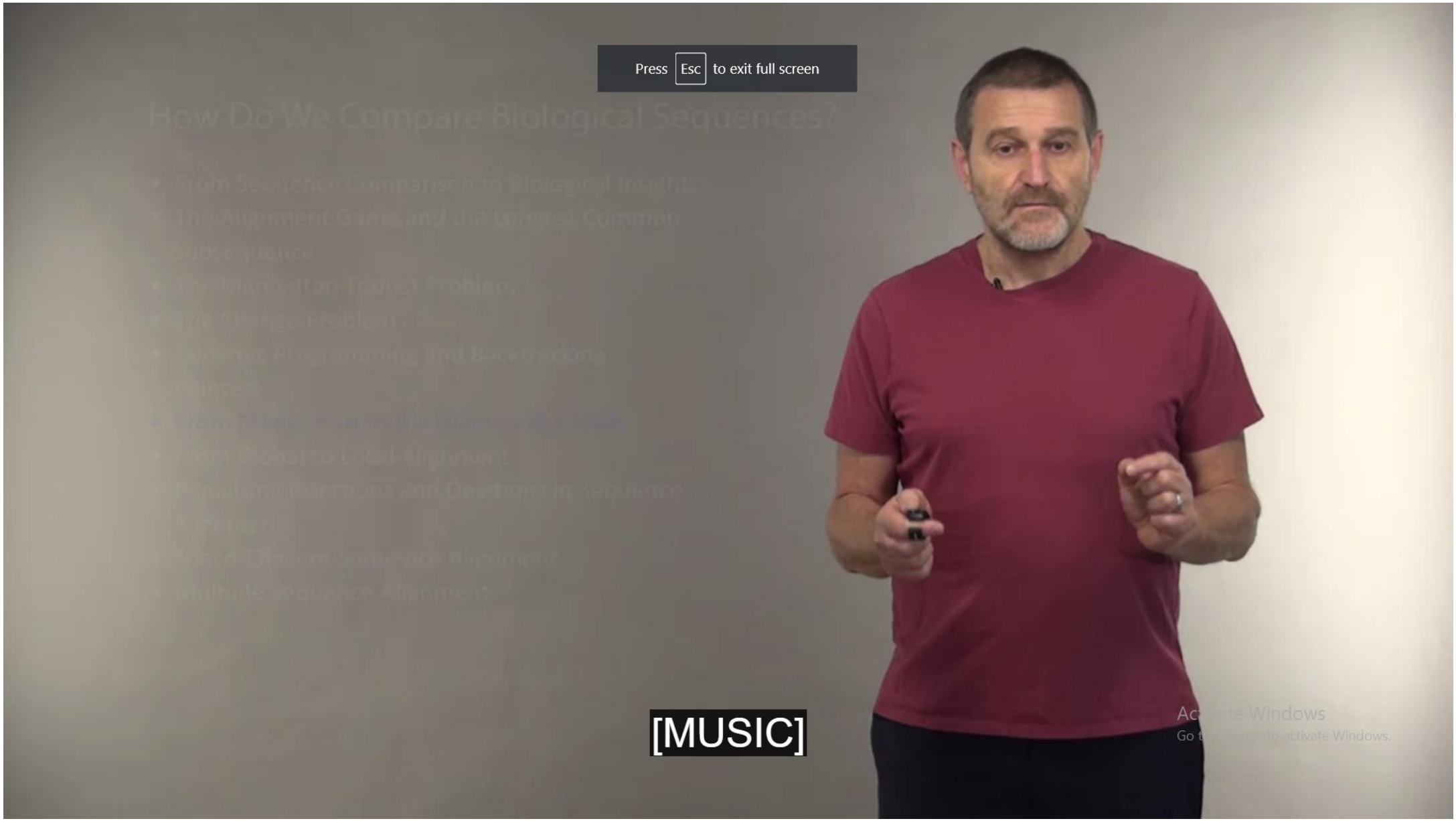


horizontal and vertical edges entering into this vertex are in bold.

What is the optimal path from *source* to *sink*?

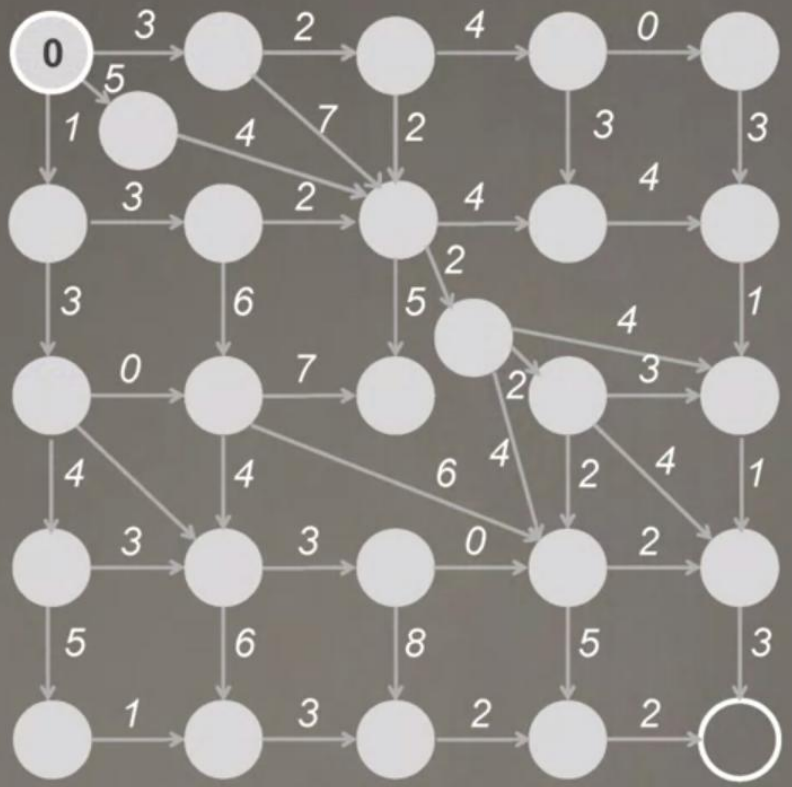


We arrive in the node labeled by 32.


Press Esc to exit full screen
How Do We Compare Biological Sequences?
[MUSIC]

Press Esc to exit full screen

How does the recurrence change for this graph?



Here's an example of a graph.

0:18 / 9:41

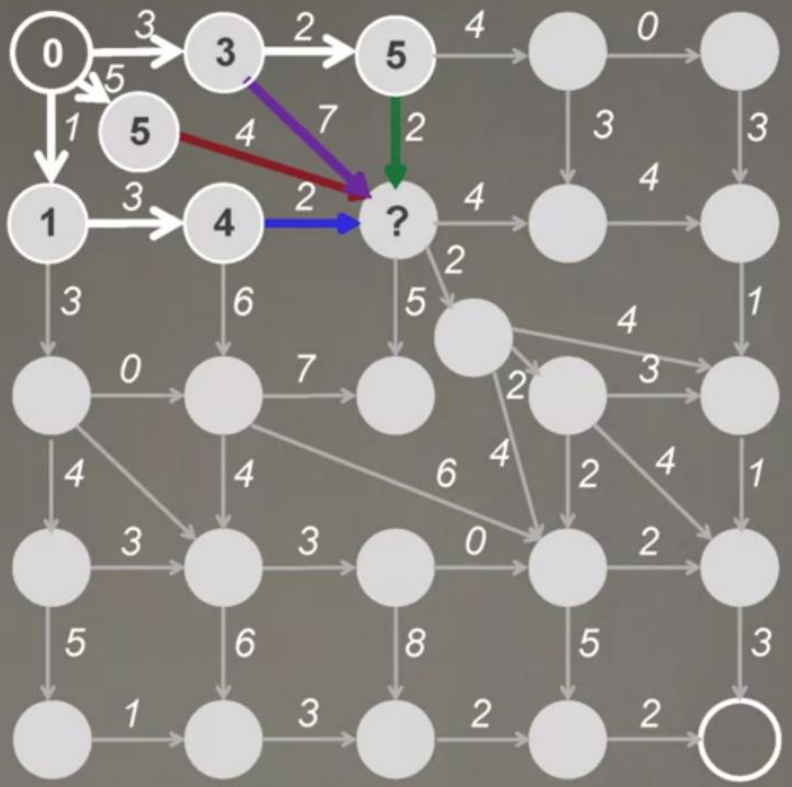
4 choices:

5 + 2

3 + 7

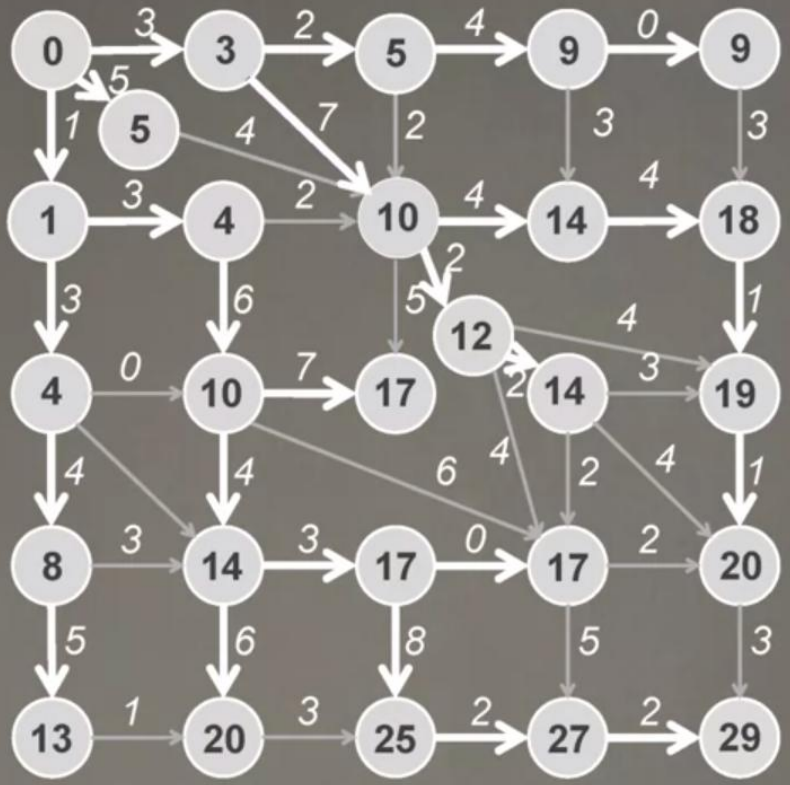
5 + 4

4 + 2



In this case, we actually have four choices.

$$s_a = \max_{\text{all predecessors } b \text{ of node } a} \{s_b + \text{weight of edge from } b \text{ to } a\}$$

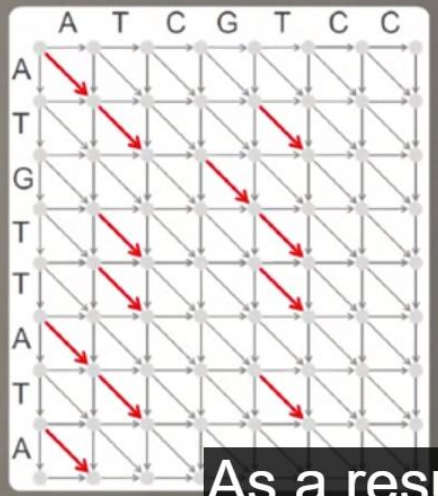


with every note marked by the length
of the longest pass and we will

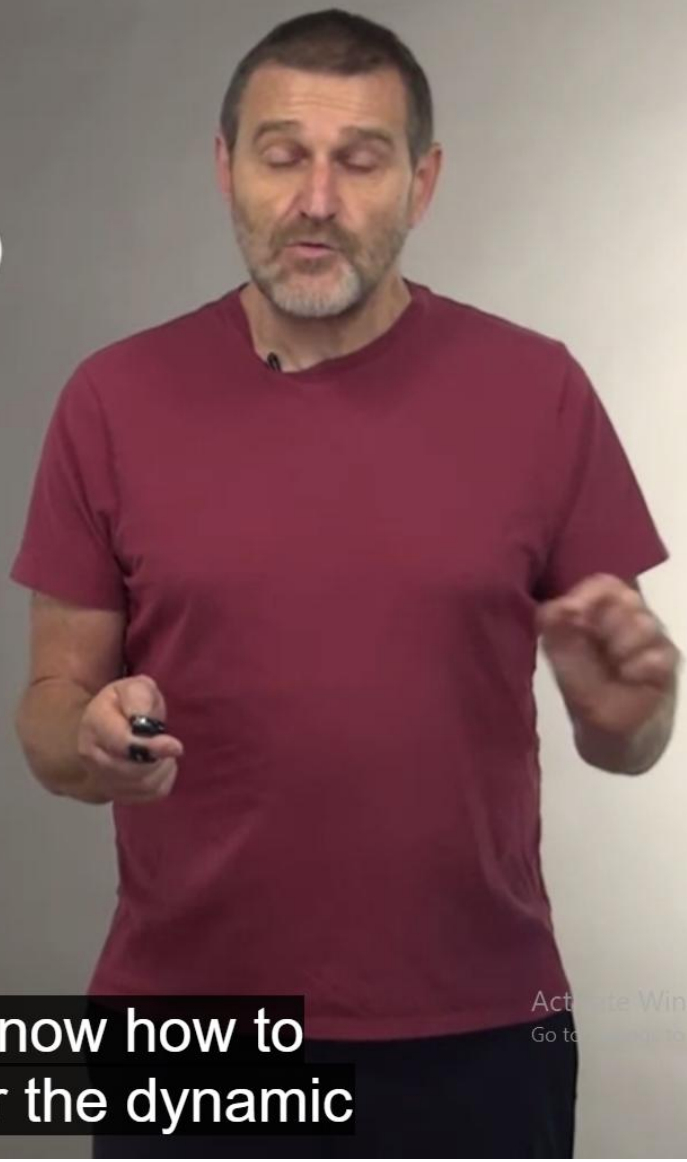
Dynamic Programming Recurrence for the Alignment Graph

$s_{i,j}$: the length of a longest path from (0,0) to (i,j)

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{weight of edge "}\downarrow\text{" into } (i,j) \\ s_{i,j-1} + \text{weight of edge "}\rightarrow\text{" into } (i,j) \\ s_{i-1,j-1} + \text{weight of edge "}\searrow\text{" into } (i,j) \end{cases}$$



red edges ↘ – weight 1
all other edges – weight 0

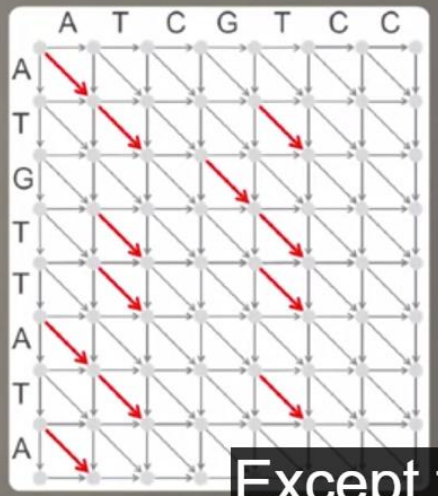


As a result, we now know how to construct recurrency for the dynamic

Dynamic Programming Recurrence for the Longest Common Subsequence Problem

$s_{i,j}$: the length of a longest path from (0,0) to (i,j)

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + 0 \\ s_{i,j-1} + 0 \\ s_{i-1,j-1} + 1, \text{ if } v_i = w_j \end{cases}$$

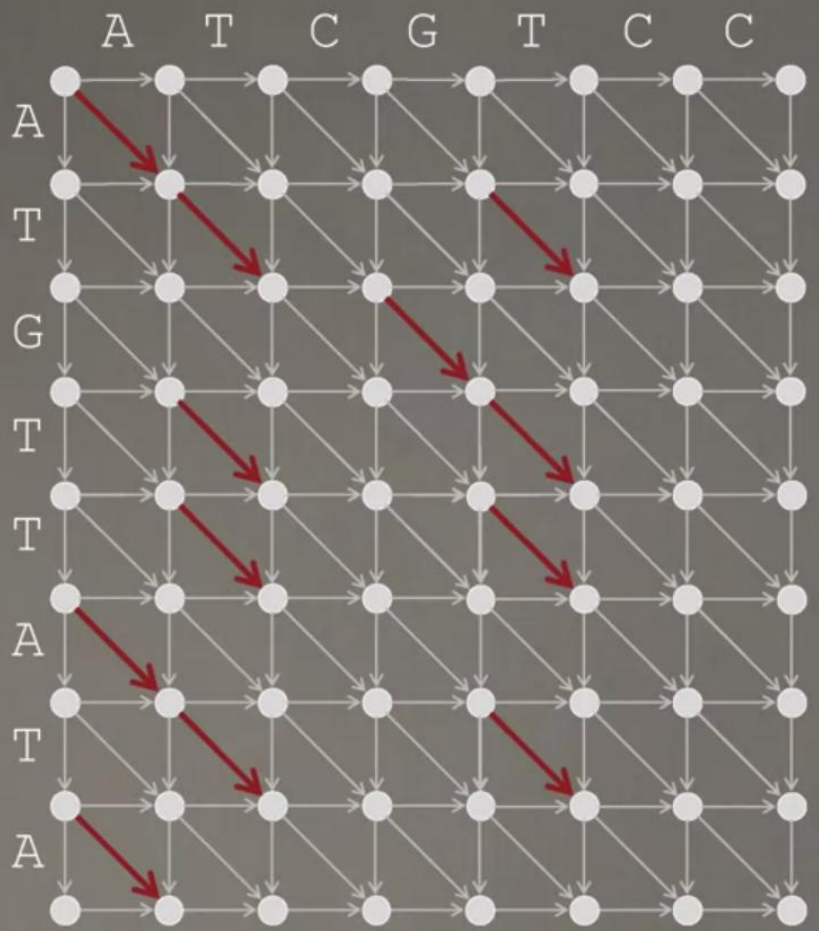


red edges ↘ – weight 1
all other edges – weight 0

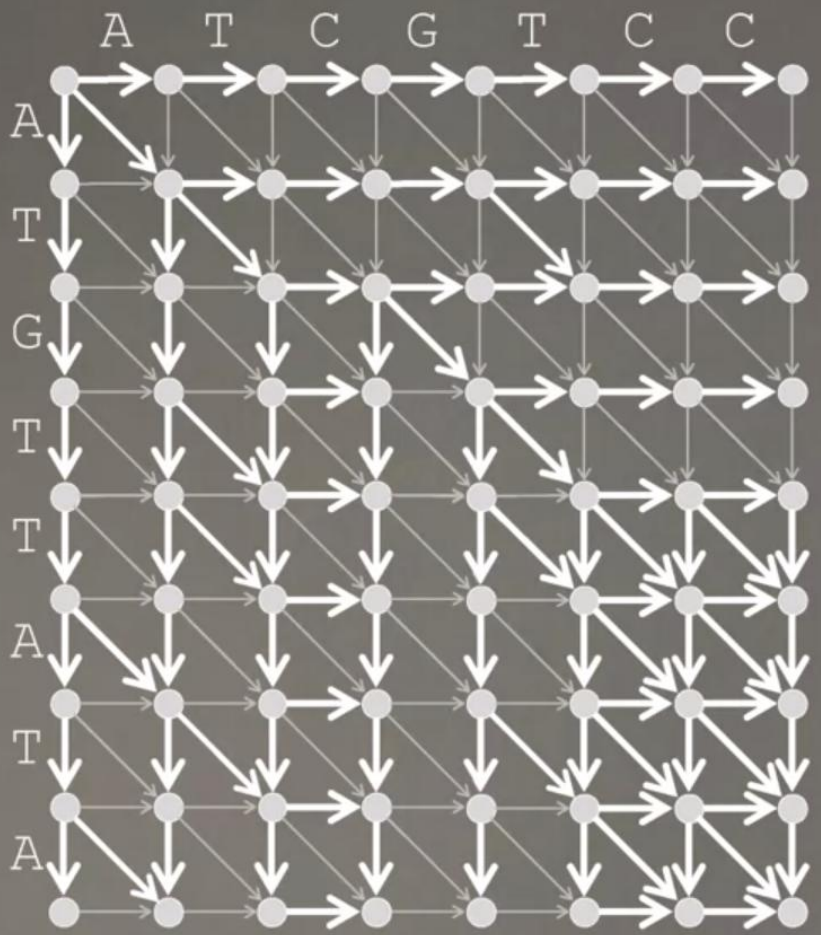
Except for the right edges, that correspond to matching letters.

backtracking pointers
for the Longest
Common Subsequence

red edges ↘ – weight 1
other edges – weight 0

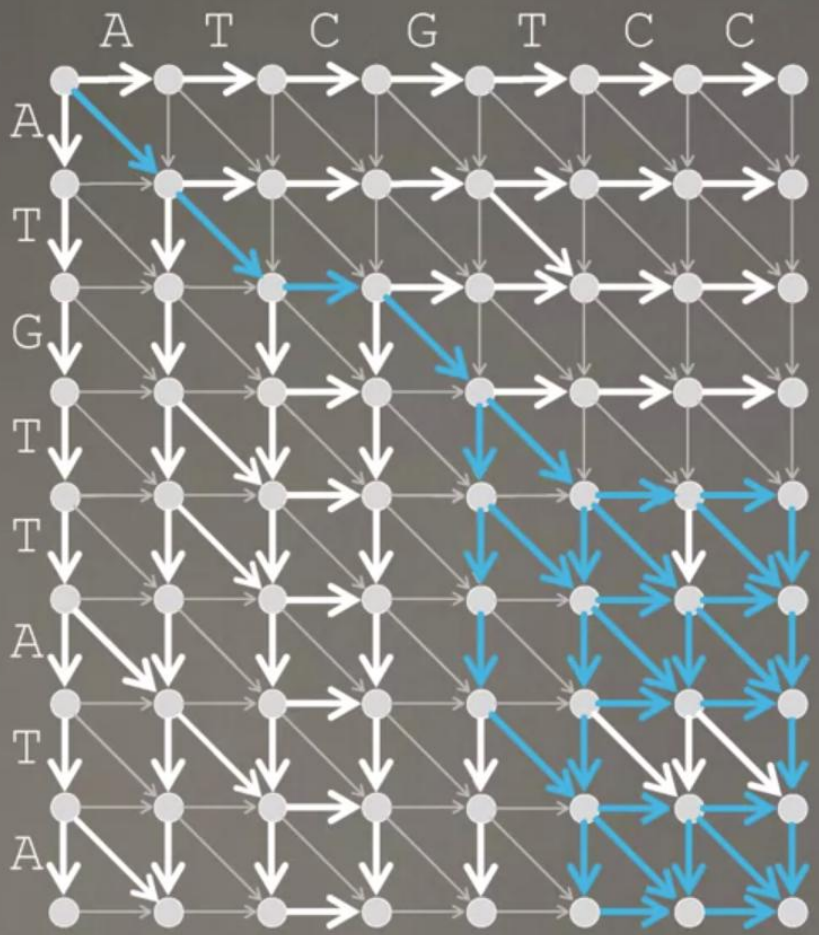


backtracking pointers
for the Longest
Common Subsequence

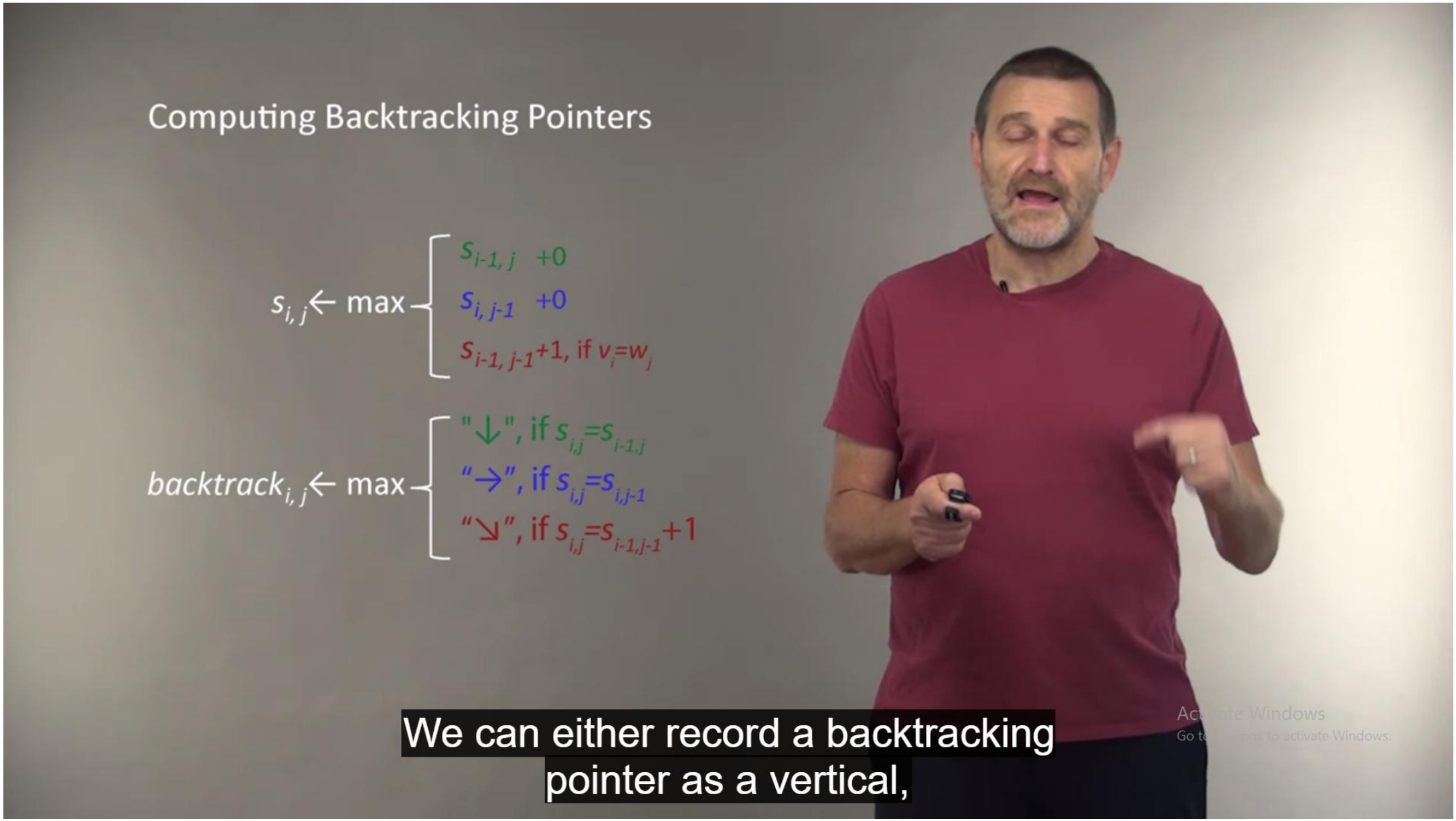


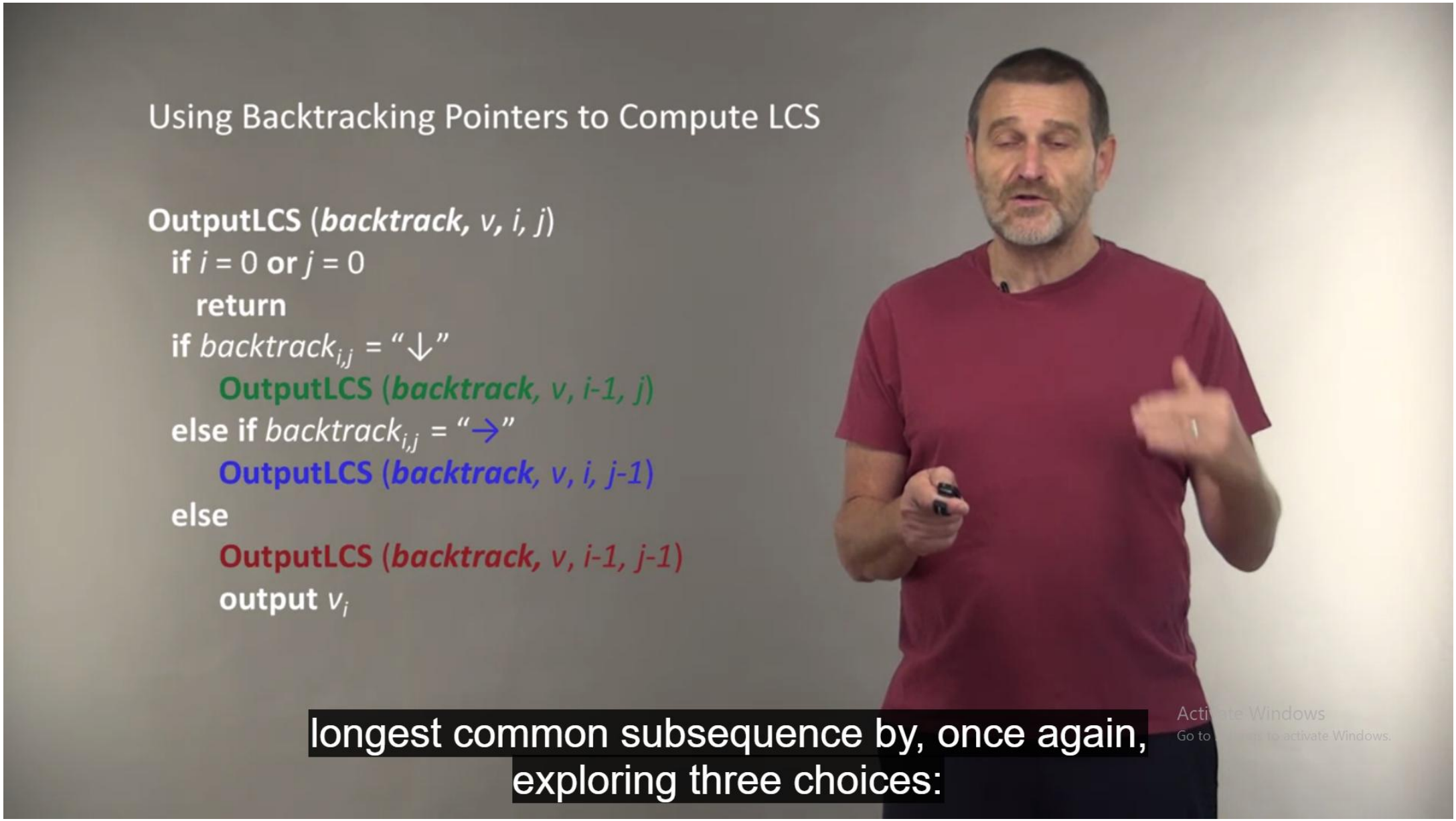
and here are the backtracking pointers for
our particular problem,

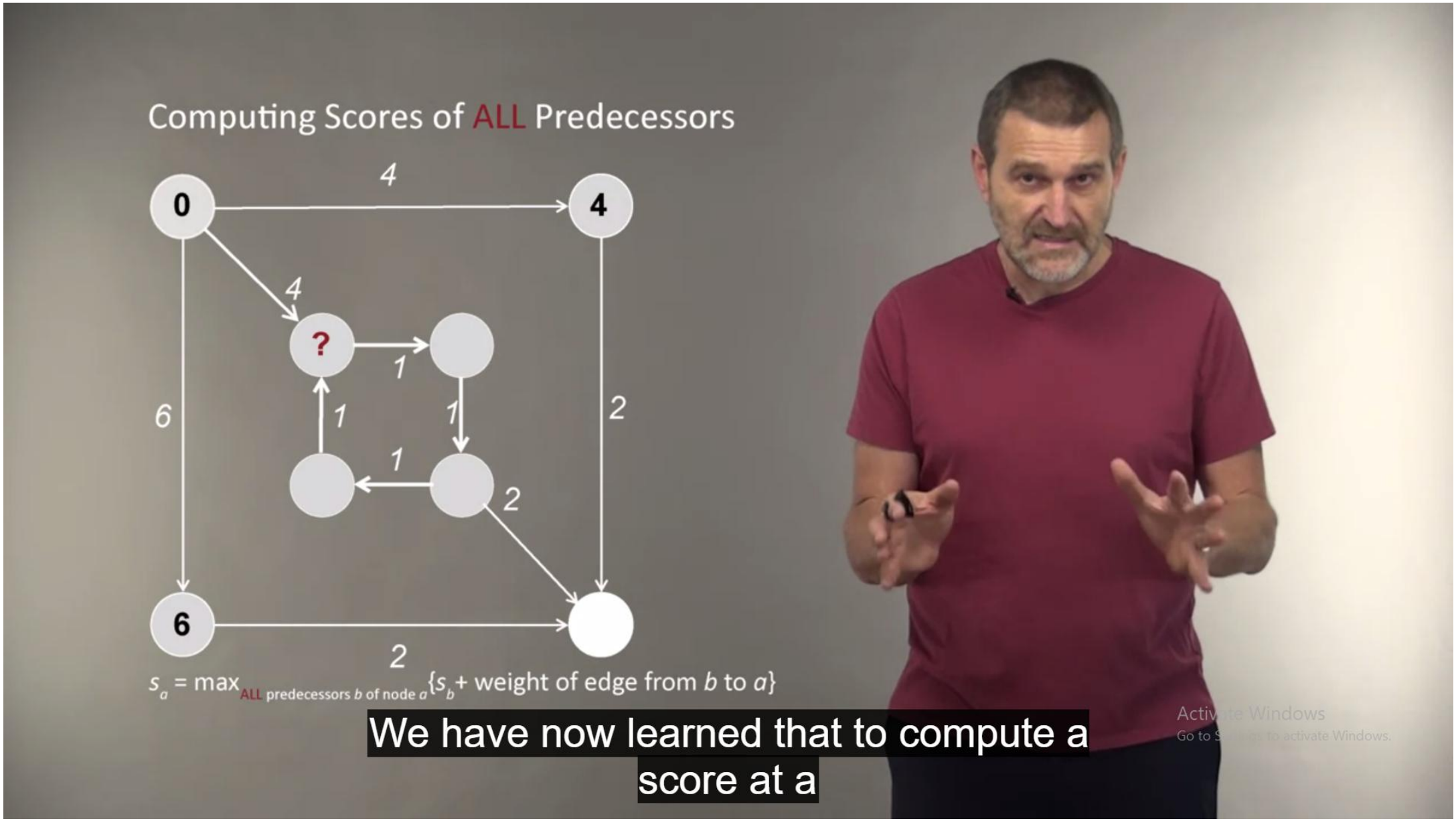
backtracking pointers
for the Longest
Common Subsequence

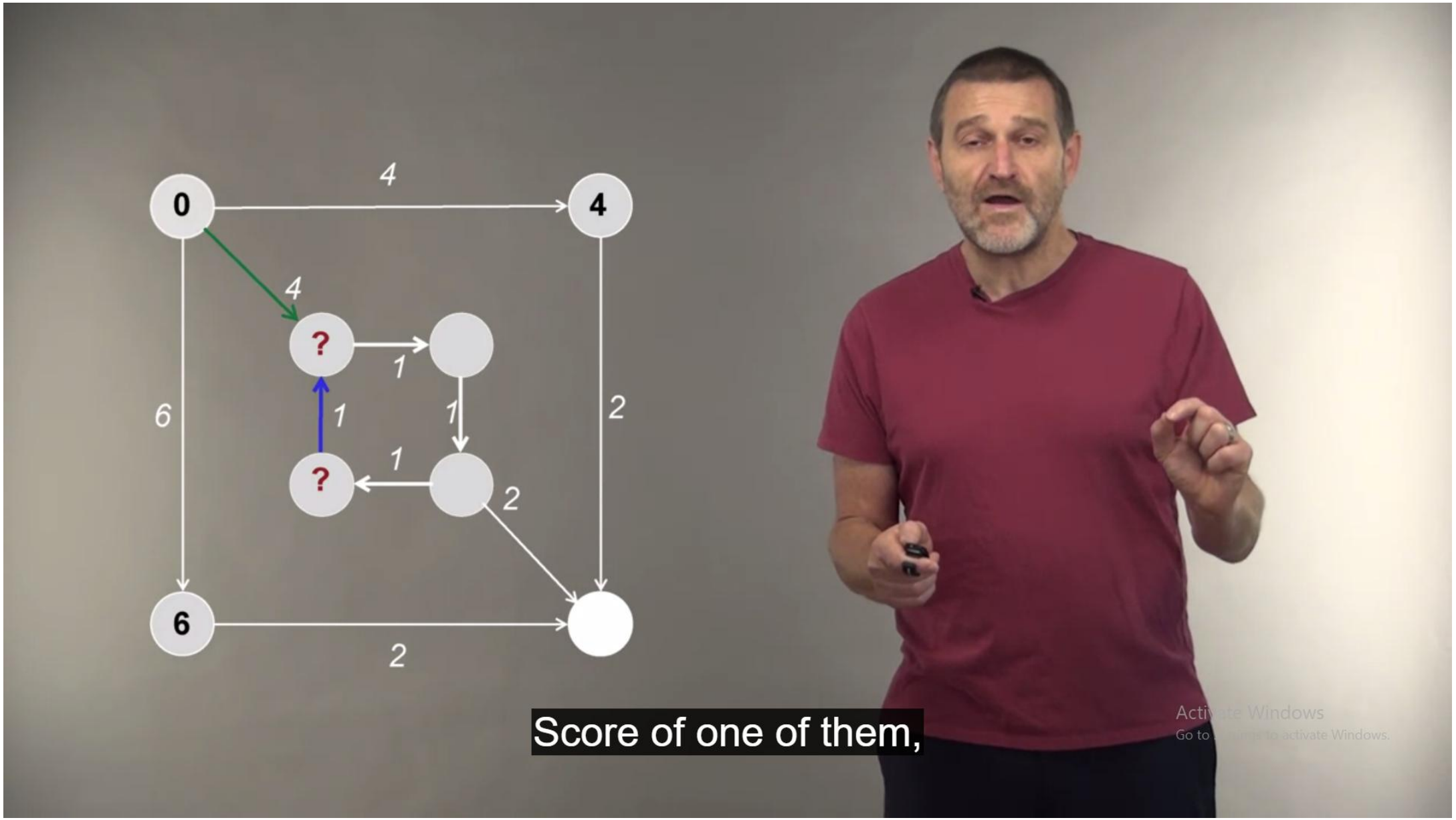


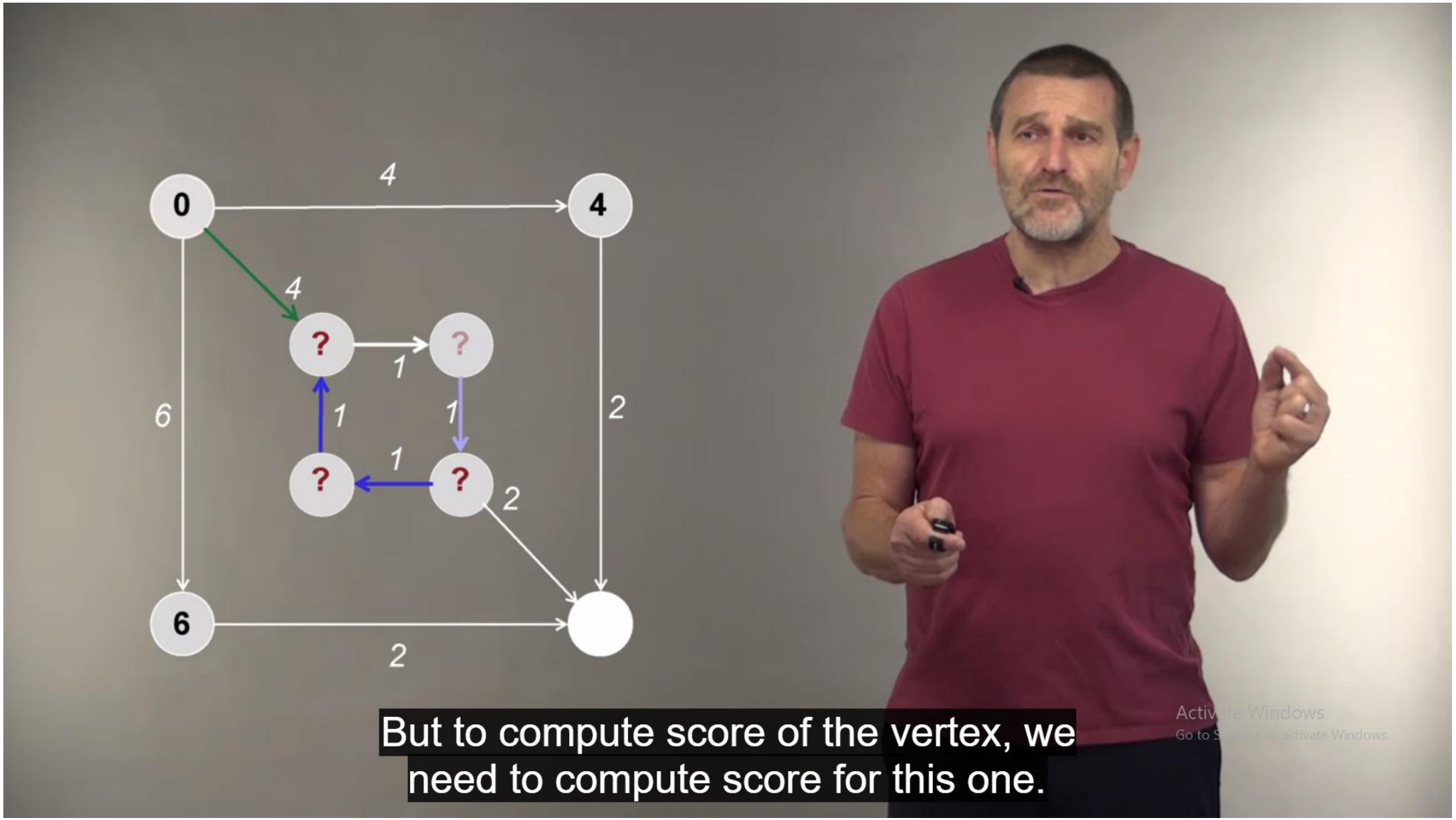
backtracking pointers, and slowly but
surely, we will arrive to


Computing Backtracking Pointers
$s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} + 0 \\ s_{i,j-1} + 0 \\ s_{i-1,j-1} + 1, \text{ if } v_i = w_j \end{cases}$
$backtrack_{i,j} \leftarrow \max \begin{cases} \text{"}\downarrow\text{"}, \text{ if } s_{i,j} = s_{i-1,j} \\ \text{"}\rightarrow\text{"}, \text{ if } s_{i,j} = s_{i,j-1} \\ \text{"}\searrow\text{"}, \text{ if } s_{i,j} = s_{i-1,j-1} + 1 \end{cases}$
We can either record a backtracking
pointer as a vertical,


Using Backtracking Pointers to Compute LCS
OutputLCS (backtrack, v, i, j)
if i = 0 or j = 0
return
if backtrack$_{i,j}$ = "↓"
OutputLCS (backtrack, v, i-1, j)
else if backtrack$_{i,j}$ = "→"
OutputLCS (backtrack, v, i, j-1)
else
OutputLCS (backtrack, v, i-1, j-1)
output v_i
longest common subsequence by, once again,
exploring three choices:
Activate Windows
Go to to activate Windows.

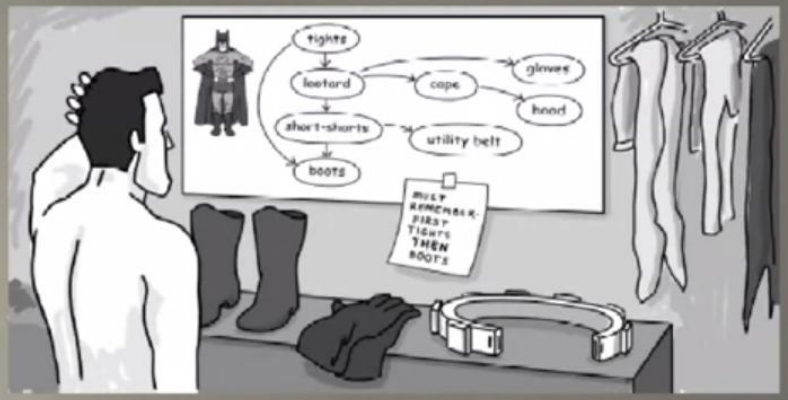

Computing Scores of ALL Predecessors
0
4
?
6
2
4
4
1
1
1
1
2
2
6
$s_a = \max_{\text{ALL predecessors } b \text{ of node } a} \{s_b + \text{weight of edge from } b \text{ to } a\}$
We have now learned that to compute a score at a


0
4
4
?
1
1
6
1
1
2
?
1
2
6
2
Score of one of them,
Activate Windows
Go to Settings to activate Windows.

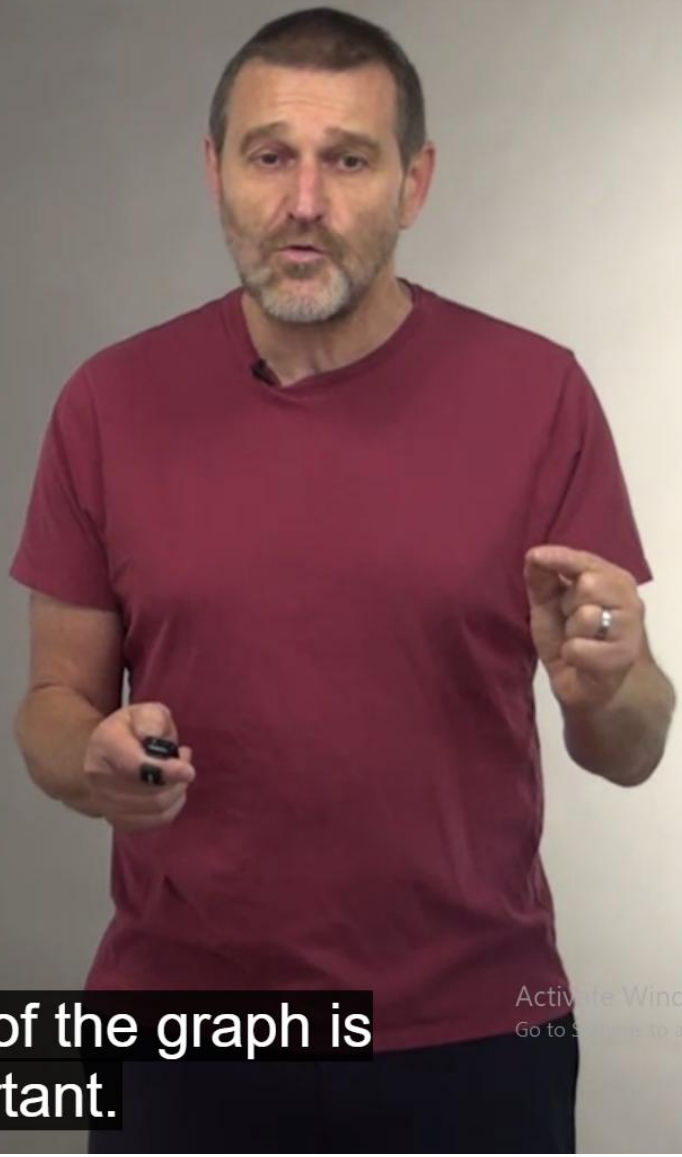

0
4
4
4
?
?
1
6
1
1
2
1
?
?
2
6
2
But to compute score of the vertex, we
need to compute score for this one.

In What Order Should We Explore Nodes of the Graph?

$s_a = \max_{\text{ALL predecessors } b \text{ of node } a} \{s_b + \text{weight of edge from } b \text{ to } a\}$



- By the time a node is analyzed, the scores of all its predecessors should already be computed
- If the graph has a **directed cycle**, this condition is impossible to satisfy
- **Directed Acyclic Graph (DAG)**: a graph without directed cycles



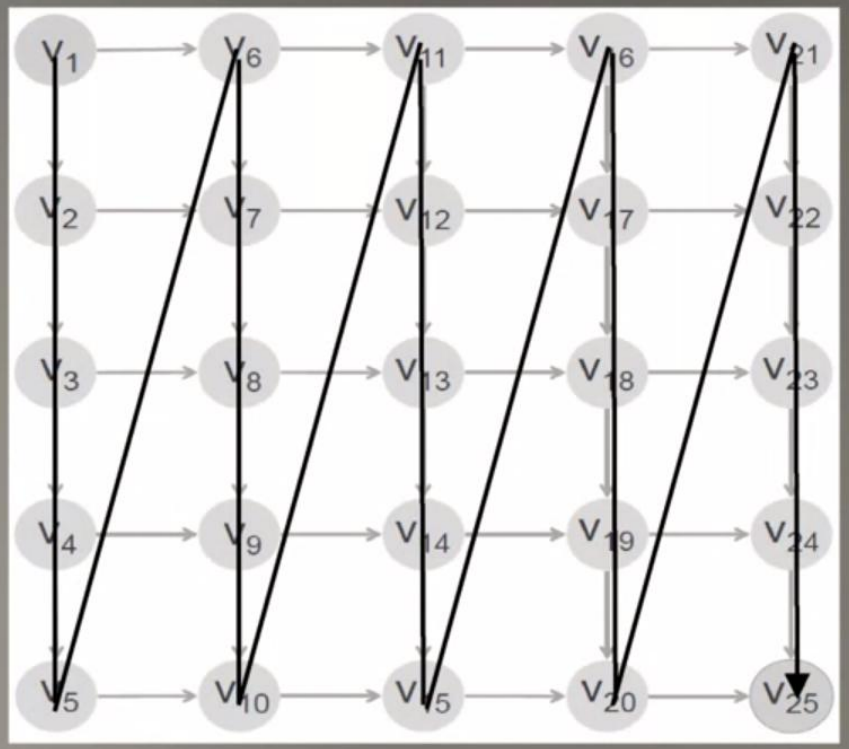
which we explore nodes of the graph is extremely important.

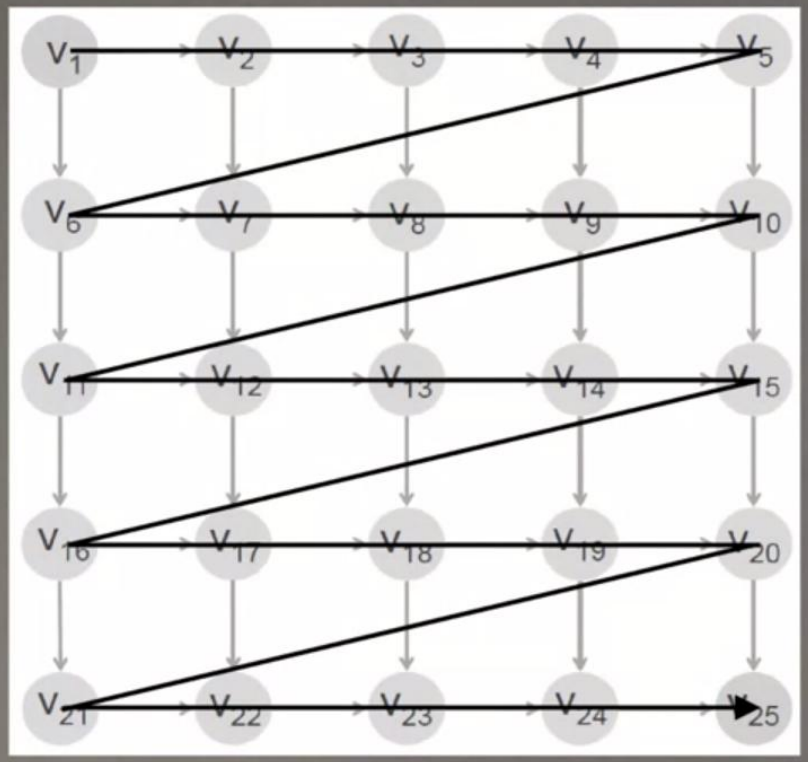
Topological Ordering

Topological Ordering: Ordering of nodes of a DAG on a line such that all edges go from left to right

And topological ordering of a graph is an ordering

Two Topological Orderings of the Manhattan Grid





And in fact, for our Manhattan grids,

LongestPath

LongestPath(*Graph, source, sink*)
  for each node *a* in *Graph*
    $s_a \leftarrow$ -infinity
  $s_{source} \leftarrow 0$
  topologically order *Graph*
  for each node *a* (from *source* to *sink* in the topological order)
    $s_a \leftarrow \max_{\text{all predecessors } b \text{ of node } a}\{s_b + \text{weight of edge from } b \text{ to } a\}$
  return s_{sink}

we can design the pseudocode for the longest path problem.

LongestPath

LongestPath(*Graph, source, sink*)
  for each node *a* in *Graph*
    $s_a \leftarrow$ -infinity
  $s_{source} \leftarrow 0$
  topologically order *Graph*
  for each node *a* (from *source* to *sink* in the topological order)
    $s_a \leftarrow \max_{\text{all predecessors } b \text{ of node } a}\{s_b + \text{weight of edge from } b \text{ to } a\}$
  return s_{sink}

runtime: O(#edges) in the graph

What is the runtime of this algorithm that we will use heavily