Solution

Overview

If an interviewer asks you this question in an interview, then their goal is *probably* to determine that:

1. You can recognize that this is a typical *shortest path* problem that can be solved with a Breadth-first search (BFS).
2. You can correctly implement a BFS to solve it.
3. For bonus points, you know that the solution could be optimized using the A\* algorithm.

For this article, I'm going to assume that you already know the fundamentals of BFS and are at the stage of figuring out how to apply it to a wide range of problems, such as this one. If you aren't yet at this stage, then I recommend checking out our relevant [Explore Card content on BFS](https://leetcode.com/explore/learn/card/queue-stack/231/practical-application-queue/1376/) before coming back to this problem.

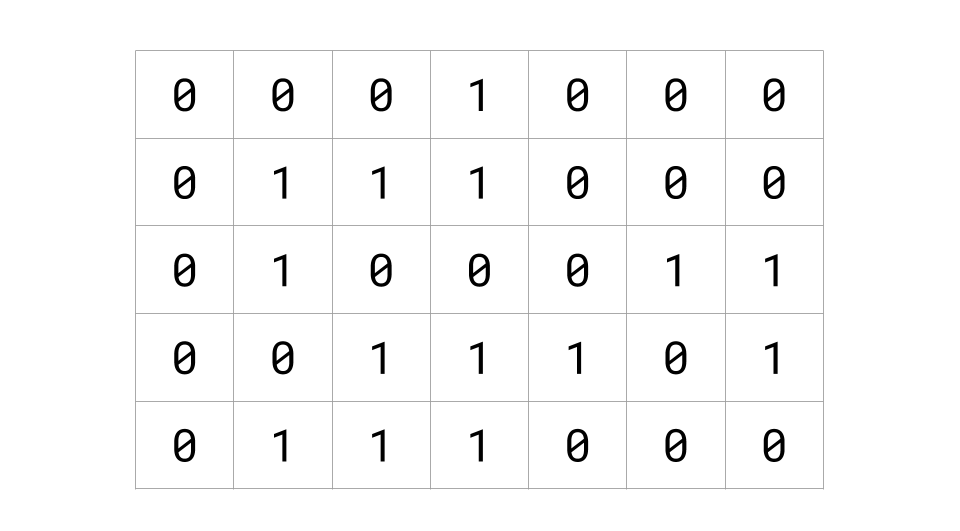
We'll look at two BFS implementations in this article; one that overwrites the input and another that does not. We'll also take a look at how this problem could be solved using A\*.

Approach 1: Breadth-first Search (BFS), Overwriting Input

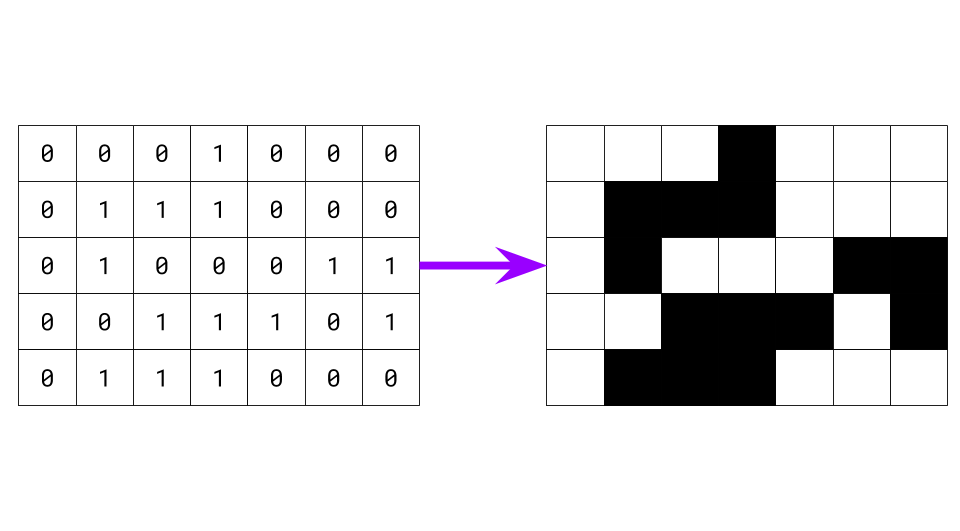
**Intuition**

*This section is aimed at readers who aren't yet at a level that they can immediately recognize this as being a BFS problem. If you don't require this level of guidance, feel free to skip forward to the algorithm or code section.*

If you're faced with a problem like this and aren't sure where to go with it, then a good first step is to make an example and solve it on a whiteboard or paper. Here's the example we'll work through here.

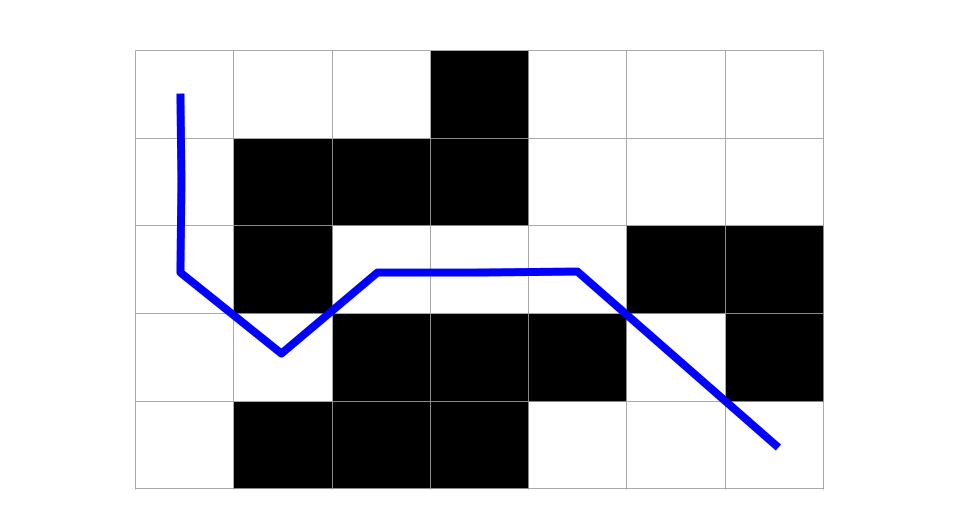


The next step is to think about whether there is a better way of visualizing the input. In this case, we're told 0's represent "open" cells, and 1's represent "blocked" cells. An intuitive way of visualizing this, therefore, is to color in the "blocked" cells.

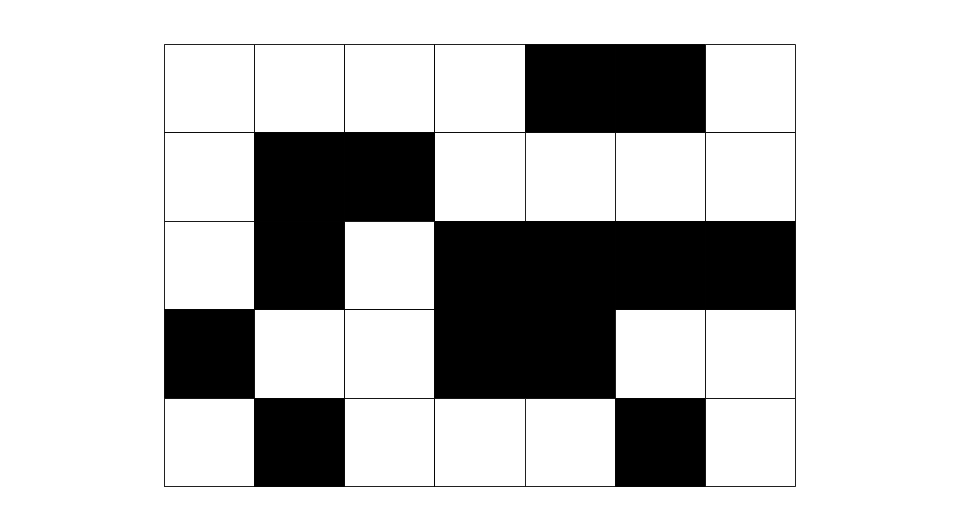


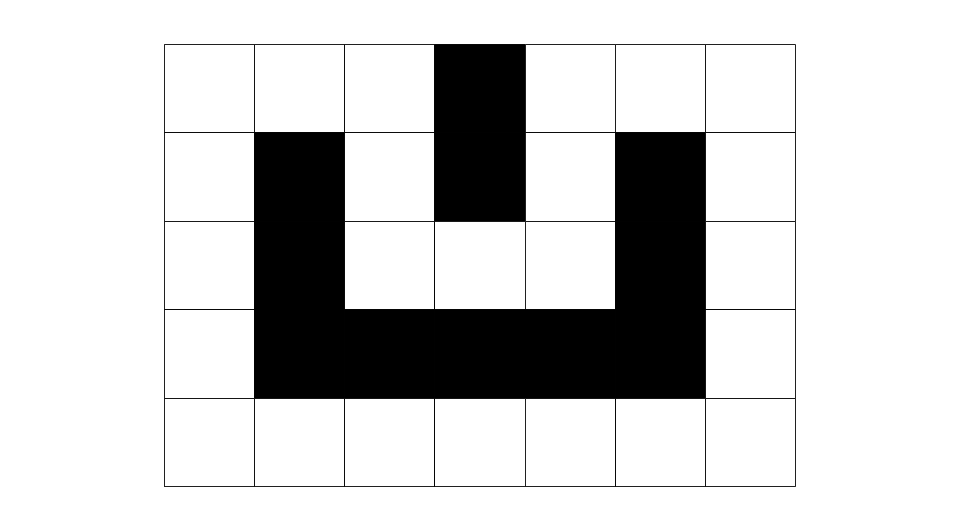
As long as you communicate clearly with your interviewer about what you're doing, making the input more "friendly" towards your eyes and brain can be one of the most effective problem-solving techniques when you're stuck. Most problems that involve grids of 0's and 1's become a lot easier when drawn like this.

Now that our example is ready to go, have a go at finding the shortest distance to get from the top-left to the bottom-right cell.



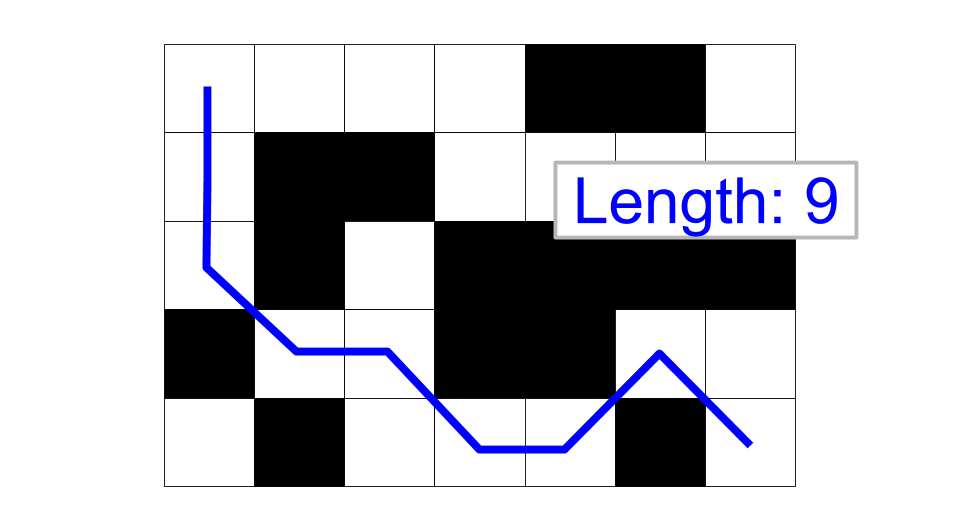
It also helps to look at more than one example. Here are a couple more for you to find the shortest distance for.

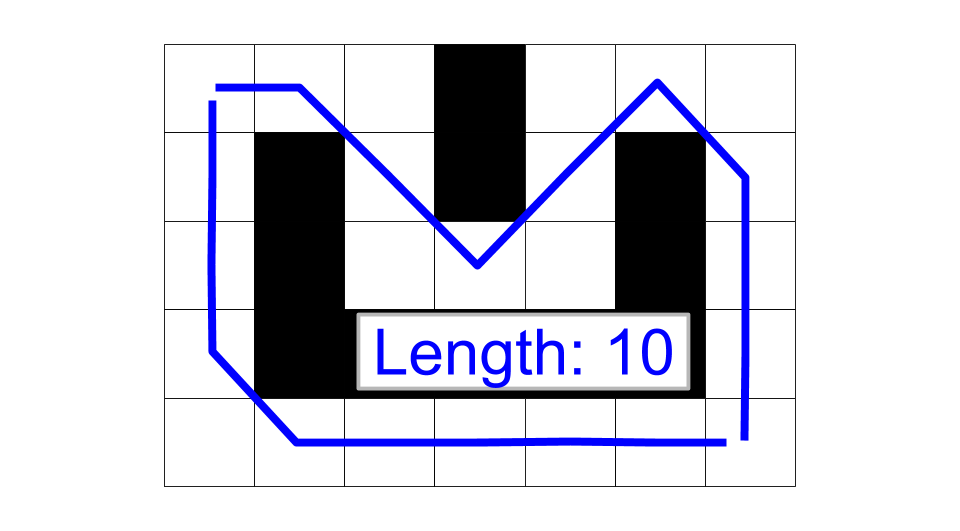




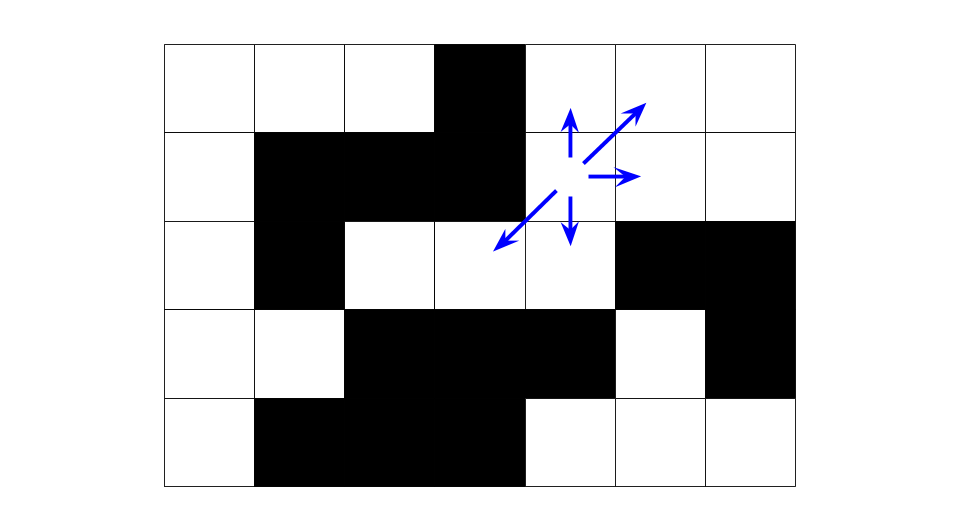
For that last one, did you find both of the shortest paths? If not, have another look!

If you did it right, these are the distances and paths you should have identified. Notice that the last one has two paths of the same length.

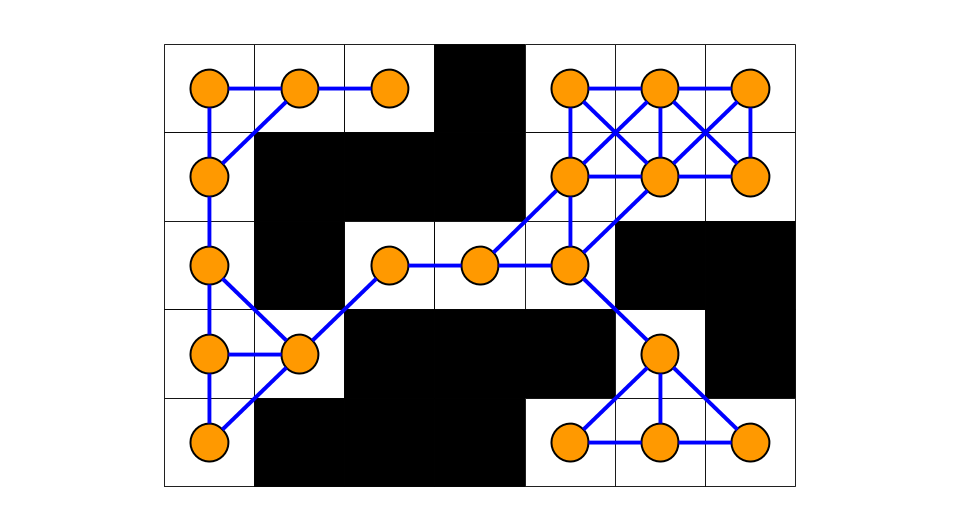




Now that we've looked at a few examples let's try and generalize a bit and come up with an algorithm. Notice that from a given cell, there are *up to* 8 options to move out of that cell into another cell. For example:



If we do this with every cell, we get the following.



Notice that what we have discovered is that the grid is a **graph**; white cells are nodes, and lines between them are edges. This is a special type of graph we call a [lattice graph](https://en.wikipedia.org/wiki/Lattice_graph). 2D arrays that are representing a graph come up a lot in interview questions. It is essential to be confident with them (don't worry too much about how we will implement this yet; we'll get to that in a bit).

So, we have a graph, and we need to find the length of the shortest path from the top-left to the bottom-right cell. Recall that to find the shortest path in a graph, we should use Breadth-first Search (BFS).

Finding the shortest path between two nodes in a graph is almost always done using BFS, and all programmers should know this. BFS is one of the fundamental algorithms that you are *expected* to be confident coding before a tech interview. So, if you're finding this question challenging, then you're doing the right thing by working on it now.

Recall that BFS works by firstly identifying all of the nodes (white cells) that can be reached in 1 step from the top-left cell, then those in 2 steps, then 3 steps, etc., until it "finds" the target node (the bottom-right cell). Here is a simple animation that shows the general way that BFS could work for this problem (this is something that you might do on the whiteboard during your interview).

1 / 17

**Algorithm**

Now that we've determined that this is a BFS problem, we need to fill in a few more details and figure out how it will all go together. Recall that BFS is implemented using a **queue**.

A queue is what we refer to as a First-In-First-Out (FIFO) data structure, comparable to people queuing to go on a theme park ride. People enter the queue at the back and leave from the front. BFS works by putting the start node on the queue, and then while the queue is non-empty, it takes a node off the *front* of the queue and puts that node's neighbors on the **back** of the queue. In this way, the graph is progressively explored, starting with the nodes nearest to the start node and ending with the nodes farthest away.

We commonly refer to putting a node on the queue as **enqueuing** and taking a node off the queue as **dequeuing**. We'll use this terminology for the remainder of the article.

Applying BFS to this problem, we'll use the queue to keep track of cells that we have numbered but haven't yet numbered the \* neighbors\* of. While usually for BFS, we'd need a "visited" set to avoid infinite looping around cycles, we won't need one for this approach because we're going to overwrite the input, and so only *unvisited* cells will have a 0 in them.

Here's the pseudocode for setting up the BFS. We identify cells with a (row, col) pair. The top-left cell is at row = 0 and col = 0 so is identified with the pair (0, 0).

queue = a new queue

enqueue cell (0, 0)

set grid[0][0] to 1

We enqueue the top-left cell as it's the first cell we'll be exploring. We also need to set its distance to be 1 in the grid (note that this will *not* cause confusion with the 1's that were used to represent blocked cells).

Now that we've done the initialization, it's time to design the main BFS loop (again, this is fairly standard template stuff).

While there are cells left on the queue, we should dequeue a cell, look up its distance (that has already been written into the input grid), and explore its neighbors. Exploring the cell's neighbors involves identifying all open cells adjacent to the current cell that *still have a 0 in them*. For each of these cells, we write the number distance + 1 into them. Finally, we need to enqueue the neighbor so that when we're ready, we can explore its neighbors too.

Here is some pseudocode.

while queue is not empty:

cell = dequeue a cell

look up distance at grid[cell row][cell col]

for each open neighbour:

if this neighbour is the bottom right cell (target):

return distance + 1

set grid[neighbour row][neighbour col] = distance + 1

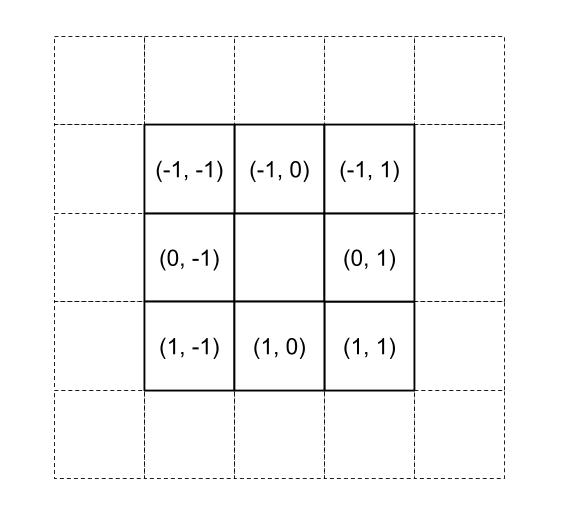
enqueue neighbour

return -1

A few points to note:

* We return -1 if the loop *terminates* without returning, as this means we ran out of cells to explore before reaching the bottom-right cell.
* The reason we can simply do distance = grid[cell row][cell col] is because cells are *only enqueued once a number has been written into them*.
* We should *only* write numbers into cells that *currently* have a 0 in them. If, for example, a cell already had a 2 in it and you then change that to a 4, it would no longer have the number that represents the *shortest* distance from the top left to itself.
* It would be okay to do the check for the bottom-right cell in the outer loop. We would need to return distance instead of distance + 1.

The final thing we need to consider is how to get all the neighbors of a cell. In traditional graph representations, this would be the equivalent of examining all the edges of a given node. For grids, we identify each neighbor by its row and column offset from the given cell.



The most common pattern is to put these "offsets" into a list as follows.

directions = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]

We can then iterate over this list and use each offset to calculate a neighbor row and column. We need to be careful, though; while most cells have 8 neighbors, corner cells only have 3 neighbors, and edges cells have 5 neighbors. To handle this, we start by checking that the neighbors row and column actually are within the dimensions of the grid. If they are within the grid, we also check that the cell currently contains a 0 (in other words, it hasn't yet been numbered and is open). If the cell contains a 0, then we add it to a list of all the neighbors to be returned.

Here is the pseudocode that puts all of this together. This function is reusable for many grid problems (usually without the 4 diagonal directions). You should be very familiar with this algorithm and be able to implement it in your programming language of choice very quickly.

directions = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]

define function get\_neighbors(row, col):

neighbors = a container to put the neighbors of (row, col) in

for each (row\_direction, col\_direction) pair in directions:

neighbor\_row = row + row\_direction

neighbor\_col = col + col\_direction

if (neighbor\_row, neighbor\_col) is NOT over the edge of the grid AND is 0:

add (neighbor\_row, neighbor\_col) to neighbors

return neighbors

Note that it is very important to check that the neighbor row and column are within the grid *before* checking the number in it. In most languages, getting this wrong will cause a crash. In Python, it will cause weird bugs due to Python's handling of negative indices.

**Code**

Here is the code for Java and Python.

class Solution:  
 def shortestPathBinaryMatrix(self, grid: List[List[int]]) -> int:  
  
 max\_row = len(grid) - 1  
 max\_col = len(grid[0]) - 1  
 directions = [  
 (-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]  
  
 # Helper function to find the neighbors of a given cell.  
 def get\_neighbours(row, col):  
 for row\_difference, col\_difference in directions:  
 new\_row = row + row\_difference  
 new\_col = col + col\_difference  
 if not (0 <= new\_row <= max\_row and 0 <= new\_col <= max\_col):  
 continue  
 if grid[new\_row][new\_col] != 0:  
 continue  
 yield (new\_row, new\_col)  
  
 # Check that the first and last cells are open.   
 if grid[0][0] != 0 or grid[max\_row][max\_col] != 0:  
 return -1  
  
 # Set up the BFS.  
 queue = deque()  
 queue.append((0, 0))  
 grid[0][0] = 1  
  
 # Carry out the BFS.  
 while queue:  
 row, col = queue.popleft()  
 distance = grid[row][col]  
 if (row, col) == (max\_row, max\_col):  
 return distance  
 for neighbour\_row, neighbour\_col in get\_neighbours(row, col):  
 grid[neighbour\_row][neighbour\_col] = distance + 1  
 queue.append((neighbour\_row, neighbour\_col))  
  
 # There was no path.  
 return -1

Some people prefer to put the logic for get\_neighbors(...) directly into the BFS loop to avoid the need for a separate function. It's a matter of personal preference as to which way is best. I prefer to keep it separate because it keeps the main BFS loop simpler and cleaner. Additionally, the logic for identifying neighbors is a separate concern to that of carrying out a BFS with them.

**Complexity Analysis**

Let N be the number of cells in the grid.

* Time complexity : O(N)*O*(*N*).

Each cell was guaranteed to be enqueued *at most once*. This is because a condition for a cell to be enqueued was that it had a zero in the grid, and when enqueuing, we also permanently changed the cell's grid value to be non-zero.

The outer loop ran as long as there were still cells in the queue, dequeuing one each time. Therefore, it ran at most N*N* times, giving a time complexity of O(N)*O*(*N*).

The inner loop iterated over the unvisited neighbors of the cell that was dequeued by the outer loop. There were at most 88 neighbors. Identifying the unvisited neighbors is an O(1)*O*(1) operation because we treat the 88 as a *constant*.

Therefore, we have a time complexity of O(N)*O*(*N*).

* Space complexity : O(N)*O*(*N*).

The only additional space we used was the queue. We determined above that at most, we enqueued N*N* cells. Therefore, an upper bound on the worst-case space complexity is O(N)*O*(*N*).

Given that BFS will have nodes of at most two unique distances on the queue at any one time, it would be reasonable to wonder if the worst-case space complexity is actually lower. But actually, it turns out that there are cases with *massive* grids where the number of cells at a single distance is proportional to N*N*. So even with cells of a single distance on the queue, in the worst case, the space needed is O(N)*O*(*N*).

Approach 2: Breadth-first Search (Without Overwriting the Input)

**Intuition**

The first approach is nice in that it's very intuitive—it's directly based on how you might solve the problem on a whiteboard. It also avoids the need for a "visited" set or data structures to keep track of distances, thus saving a constant amount of memory over typical BFS implementations. *However*, like all in-place algorithms, overwriting the input can cause problems. Here are a couple of possible scenarios you need to consider.

1. That the algorithm is running in a \* multithreaded\* environment, and it does not have exclusive access to the grid. Other threads might need to read the grid too, and might not expect it to be modified.
2. That there is only a single thread *or* the algorithm has exclusive access to the grid while running, **but** the grid might need to be reused later or by another thread once the lock has been released.

For the second scenario, Approach 1 could be modified to restore the grid—simply loop over it at the end, replacing all numbers that are greater than 1 with a 0, and additionally set the top-left cell to a 0.

For the first scenario, though, this won't work. We would have to come up with an algorithm that doesn't modify the input.

You should **always** discuss the possibility of overwriting the input with your interviewer and clarify what kind of environment your algorithm is expected to run in. Sometimes they won't care, sometimes they'll state it has to run in a multithreaded environment, or sometimes they'll have a particular preference as it impacts what they're trying to see from you.

Anyway, to avoid over-writing the input, we could go for a more traditional BFS algorithm that uses a "visited" set (which can be implemented as either a hash set or a new 2D array). It will also need to keep track of the distances in some other way.

There are several variants of this approach that we'll look at. All are based on the standard BFS templates that you should be very familiar with (or be working towards being very familiar with).

**Algorithm**

Firstly, we can reuse our get\_neighbors(...) function from above. It won't need any modifications as we'll handle the "visited" logic in the main function. This is another big advantage of keeping the get\_neighbors(...) logic separate; even if you end up completely changing your main algorithm, this bit won't need re-writing.

For the actual BFS algorithm, there are several variants that we'll now talk about. All use a visited set. All achieve the same result and have the same time and space complexities; however, all are based on different ways of intuitively viewing the problem. Like always, it's great to understand more than one way of solving a problem.

*Distances on queue*

The simplest variant is to use a visited set and to put the distances on the queue, alongside the row and column (triplets instead of pairs).

visited = a new set

queue = a new queue

enqueue (0, 0, 1)

add (0, 0) to visited

while the queue is not empty:

row, col, distance = dequeue and unpack a cell

if (row, col) is the bottom right cell:

return distance

for each open neighbour:

if neighbour is in visited:

continue

otherwise, add neighbour to visited

enqueue (neighbour\_row, neighbour\_col, distance + 1)

return -1

This approach is nice in that it's very easy to code and reason about. While the distances going on the queue are repetitive, correct code is generally worth a lot more "points" in an interview than attempted-to-optimize-but-it' s-buggy code is!

*Starting a new collection for each distance*

BFS works by examining cells in order of distance from the starting point. In other words, all cells at a distance of x are visited before any cells at a distance of x + 1. Additionally, cells at a distance of x can only enqueue other cells that are at a distance of x + 1. Therefore, there are at most two unique distances in the queue at any one time.

This implementation utilizes this property by keeping track of two collections of cells to be explored: the remaining ones at the current distance and the ones at the next distance. There is no need to use queues, as the order that cells of the same distance are explored does not matter. Any data structure that has O(1)*O*(1) insertions and removals will do.

visited = a new set

add (0, 0) to visited

current\_layer = a new list

next\_layer = a new list

add (0, 0) to current layer

while current\_layer is not empty:

row, col = remove and unpack a cell from current\_layer

if (row, col) is the bottom right cell (target):

return distance

for each open neighbor:

if the neighbor is in visited:

continue

add neighbor to visited

add neighbor to next\_layer

if current\_layer is now empty:

current\_distance += 1

current\_layer = next\_layer

next\_layer = a new list

return -1

*Keeping track of how many cells at each distance are on the queue*

This final implementation uses the same BFS property as the above one, but in a different way.

At the start, there is exactly 1 cell at a distance of 1. Once we have dequeued and processed that cell, we know all cells currently in the queue must be of distance 2. We can check at this point how many of them there are and then dequeue and process that number of cells. Now we know all of the cells in the queue are of distance 3. This argument extends to the entire grid.

visited = a new set

queue = a new queue

enqueue cell (0, 0)

add (0, 0) to visited

current\_distance = 1

while queue is not empty:

nodes\_on\_queue = current queue length

repeat nodes\_on\_queue\_times:

row, col = dequeue and unpack a cell

if (row, col) is the bottom right cell (target):

return distance

for each open neighbour:

if neighbour is in visited:

continue

add neighbour to visited

enqueue (neighbour\_row, neighbour\_col, distance + 1)

current\_distance += 1

return -1

While elegant, this implementation is more complicated and has more room for bugs. If you are confident, though, this is probably the way to go.

**Code**

In Python, it is idiomatic to use a set to keep track of visited cells. In Java, you could also use a HashSet. However, a 2D array is possibly a better option for performance and is a bit less of a headache to implement. The implementations provided here use a 2D array for Java.

I've provided code for all 3 sub-implementations I discussed in the previous section.

*Distances on queue*

class Solution:  
 def shortestPathBinaryMatrix(self, grid: List[List[int]]) -> int:  
  
 max\_row = len(grid) - 1  
 max\_col = len(grid[0]) - 1  
 directions = [  
 (-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]  
  
 # Helper function to find the neighbors of a given cell.  
 def get\_neighbours(row, col):  
 for row\_difference, col\_difference in directions:  
 new\_row = row + row\_difference  
 new\_col = col + col\_difference  
 if not (0 <= new\_row <= max\_row and 0 <= new\_col <= max\_col):  
 continue  
 if grid[new\_row][new\_col] != 0:  
 continue  
 yield (new\_row, new\_col)  
  
 # Check that the first and last cells are open.   
 if grid[0][0] != 0 or grid[max\_row][max\_col] != 0:  
 return -1  
  
 # Set up the BFS.  
 queue = deque([(0, 0, 1)])  
 visited = {(0, 0)}  
  
 # Do the BFS.  
 while queue:  
 row, col, distance = queue.popleft()  
 if (row, col) == (max\_row, max\_col):  
 return distance  
 for neighbour in get\_neighbours(row, col):  
 if neighbour in visited:  
 continue  
 visited.add(neighbour)  
 # Note that the \* splits neighbour into its values.  
 queue.append((\*neighbour, distance + 1))  
  
 # There was no path.  
 return -1

*Starting a new collection for each distance*

class Solution:  
 def shortestPathBinaryMatrix(self, grid: List[List[int]]) -> int:  
  
 max\_row = len(grid) - 1  
 max\_col = len(grid[0]) - 1  
 directions = [  
 (-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]  
  
 # Helper function to find the neighbors of a given cell.  
 def get\_neighbours(row, col):  
 for row\_difference, col\_difference in directions:  
 new\_row = row + row\_difference  
 new\_col = col + col\_difference  
 if not (0 <= new\_row <= max\_row and 0 <= new\_col <= max\_col):  
 continue  
 if grid[new\_row][new\_col] != 0:  
 continue  
 yield (new\_row, new\_col)  
  
 # Check that the first and last cells are open.   
 if grid[0][0] != 0 or grid[max\_row][max\_col] != 0:  
 return -1  
  
 # Set up the BFS.  
 current\_layer = [(0, 0)]  
 next\_layer = []  
 visited = {(0, 0)}  
 current\_distance = 1  
  
 while current\_layer:  
  
 # Process the current layer.  
 for row, col in current\_layer:  
 if (row, col) == (max\_row, max\_col):  
 return current\_distance  
 for neighbour in get\_neighbours(row, col):  
 if neighbour in visited:  
 continue  
 visited.add(neighbour)  
 next\_layer.append(neighbour)  
  
 # Set up for processing the next layer.  
 current\_distance += 1  
 current\_layer = next\_layer  
 next\_layer = []  
  
 # There was no path.  
 return -1

*Keeping track of how many cells at each distance are on the queue*

class Solution:  
  
 def shortestPathBinaryMatrix(self, grid: List[List[int]]) -> int:  
  
 max\_row = len(grid) - 1  
 max\_col = len(grid[0]) - 1  
 directions = [  
 (-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]  
  
 # Helper function to find the neighbors of a given cell.  
 def get\_neighbours(row, col):  
 for row\_difference, col\_difference in directions:  
 new\_row = row + row\_difference  
 new\_col = col + col\_difference  
 if not (0 <= new\_row <= max\_row and 0 <= new\_col <= max\_col):  
 continue  
 if grid[new\_row][new\_col] != 0:  
 continue  
 yield (new\_row, new\_col)  
  
 # Check that the first and last cells are open.   
 if grid[0][0] != 0 or grid[max\_row][max\_col] != 0:  
 return -1  
  
 # Set up the BFS.  
 queue = deque([(0, 0)])  
 visited = {(0, 0)}  
 current\_distance = 1  
  
 # Do the BFS.  
 while queue:  
 # Process all nodes at current\_distance from the top-left cell.  
 nodes\_of\_current\_distance = len(queue)  
 for \_ in range(nodes\_of\_current\_distance):  
 row, col = queue.popleft()  
 if (row, col) == (max\_row, max\_col):  
 return current\_distance  
 for neighbour in get\_neighbours(row, col):  
 if neighbour in visited:  
 continue  
 visited.add(neighbour)  
 queue.append(neighbour)  
 # We'll now be processing all nodes at current\_distance + 1  
 current\_distance += 1  
  
 # There was no path.  
 return -1

**Complexity Analysis**

Let N be the number of cells in the grid.

* Time complexity : O(N)*O*(*N*).

Same as approach 1. Processing a cell is O(1)*O*(1), and each of the N*N* cells is processed at most once, giving a total of O(N)*O*(*N*).

* Space complexity : O(N)*O*(*N*).

Same as approach 1. The visited set also requires O(N)*O*(*N*) space; in the worst case, it will hold the row and column of each of the N*N* cells.

Approach 3: A\* (Advanced)

**Intuition**

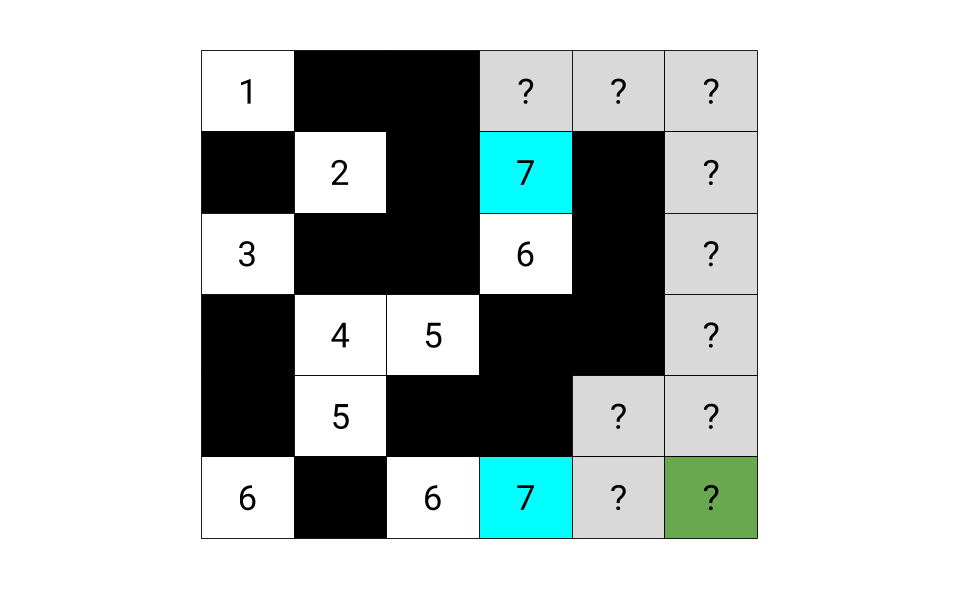
For this problem, A\* has a slightly worse time complexity, but in practice, performs quite well. It is a lot more complicated to code, and on problems such as this one, it doesn't offer a clear benefit. In saying that, it's worth knowing about as it allows you to show the depth of your ability while discussing the pros and cons of your BFS implementation.

In an interview, you need to be a bit careful, though. Going straight to A\* could make you come across as somebody who tends to [overengineer](https://en.wikipedia.org/wiki/Overengineering) code. After all, there is nothing wrong with the BFS approach, and A\* is a lot more complicated. A\* is also extremely risky to code in an interview, as the implementation details can be quite challenging to get right.

So, feel free to briefly mention A\* as a possibility in the initial "approaches" discussion, but unless the interviewer encourages you to pursue it, then you should point out that BFS is a far simpler approach, ideal for this particular problem. If you coded your BFS very quickly, then explaining and possibly implementing A\* is a *possible* follow up question you could be asked.

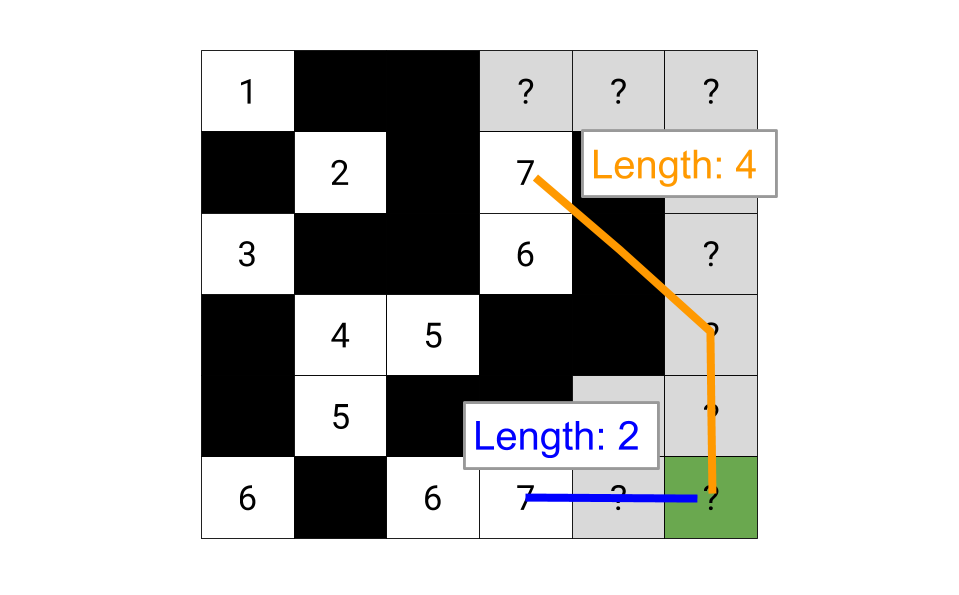
Anyway, now that we've talked about how A\* could fit into an interview for this problem let's get started on learning what it is! In approaches 1 and 2, we used a min-queue to keep track of cells we still needed to explore the neighbors of. The use of a min-queue ensured that we **always** explored the cells that we had traveled the smallest distance so far to get to. This meant that when we found the bottom-right cell, we knew we must have traveled the *minimum possible* distance to get to it.

You might have noticed that this approach somewhat lacked "common sense", though. For example, here is the state of a grid, partway through Approach 2.



At this point, one of the two cells to explore seemed far more promising than the other. We want to modify the algorithm to prioritize promising paths over not so promising paths. To do this, we need to come up with a **heuristic** that, given a potential option, it measures how much "promise" that option has. Then we prioritize the options (exploring cells) with the highest "promise". In the previous approaches, we were actually doing this; our heuristic was simply *distance traveled so far*. But we can do better than that!

Instead, we're going to score them based on distance traveled so far **plus** our most **optimistic** estimate of how many more steps it would take to get to the bottom-right cell from there. For our example above, these estimates would be as follows. Indeed, this new heuristic favors the option that looks better.



Mathematically, this optimistic estimate for the remainder of the path is simply the maximum out of the number of rows and columns remaining (intuitively, we can always cover the smaller distance within the larger one by moving diagonally).

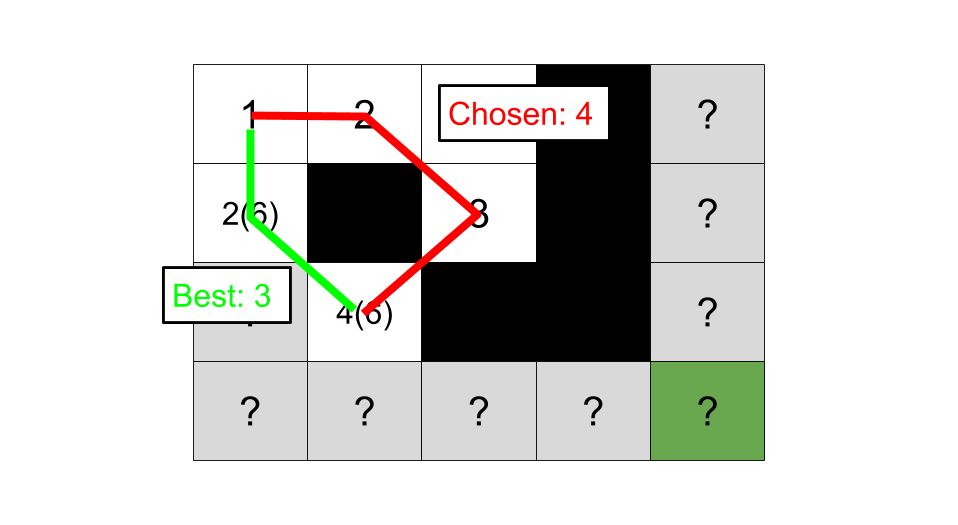
Here is an animation of *attempting* to modify one of our earlier approaches to use this new heuristic. As a **warning**, though, this exact approach \*\* doesn't work in all cases\*\*, and we'll be making a tweak to it shortly. We're looking at it anyway as a starting point, though, as a lot of people are likely to initially come up with it, and so it really illustrates the level of caution needed when implementing A\*!

1 / 21

If you weren't warned beforehand, you might have believed this algorithm was fine. In fact, at the time of writing, it actually passed 64/84 test cases here on LeetCode! Here is an animation of a case it fails on, though. Try to spot exactly where it started going wrong.

1 / 17

So, what's going on? We have made the wrong assumption that we can take the most promising cell and assume that the best possible distance to all of its neighbors is 1 more. But this is not always the case. In the above example, the most promising looking path went back up to a cell that could have been better accessed by initially taking a not so great looking path.



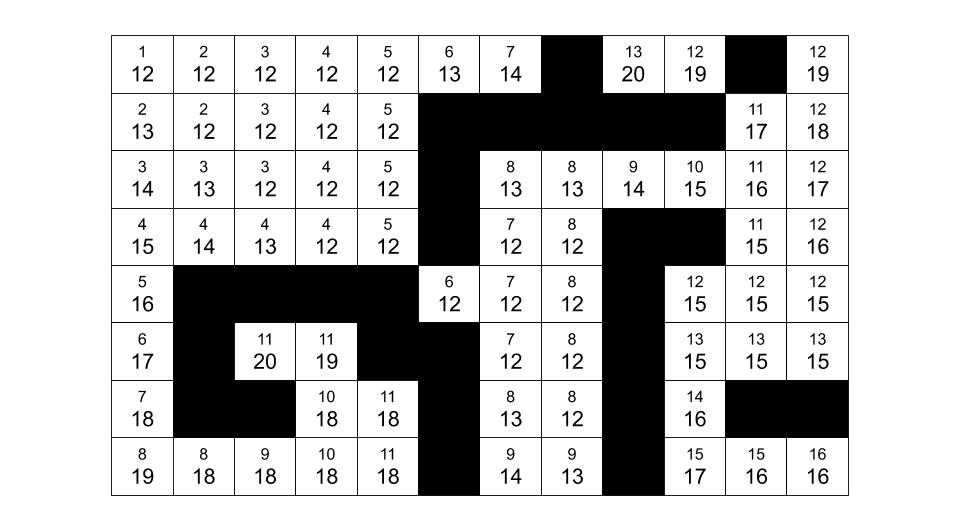
To solve this, we simply need to remove the bad assumption; we shouldn't conclude that the first path we discover into a cell is the best one. Instead, we should keep track of all the options and then choose the best one when we get to it. Here is an animation of this improved algorithm.

1 / 17

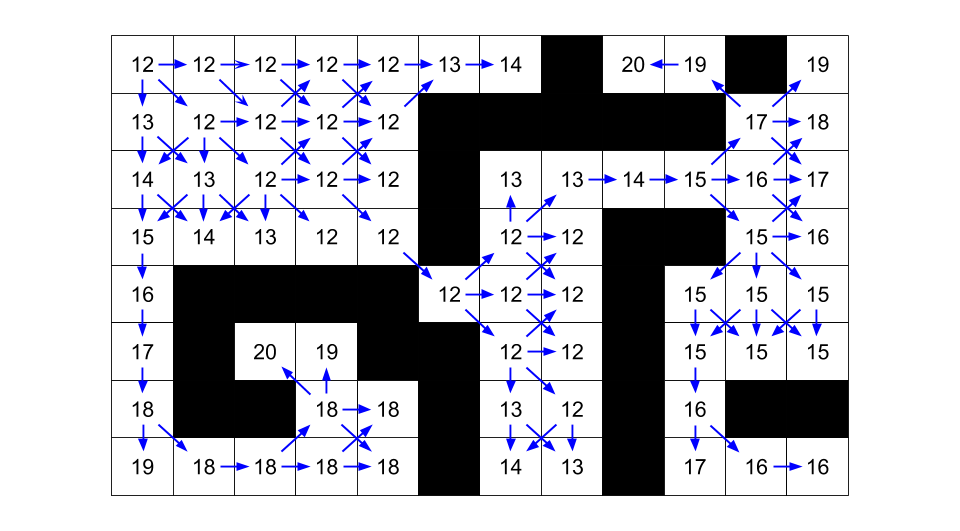
We'll now take a closer look at what exactly A\* is doing to help you build some intuition around it. After that, we'll formally prove that this new heuristic is indeed guaranteed to give an optimal answer.

The estimates we wrote into cells represented the shortest *possible* path we could get from the top-left to the bottom-right if we were to go through that cell. This was guaranteed, as we had identified the shortest possible path to get to that cell from the top-left, and we knew that after that, the path could make at most 1 move right and 1 move down in a single step. It was simply impossible that it could be less than that. The actual best path that goes through this cell could be *longer* than the estimate, but not shorter.

Here is an image showing estimates for all the cells in a large grid. Note that A\* doesn't necessarily need to find estimates for all the cells; however, this is what it would do if we allowed it to run until the priority queue was empty, as opposed to until it had found the bottom-right cell. The smaller numbers are the distances and are included to help you relate it back to BFS.



And here is the same grid, except with arrows showing where each cell was entered from. Did you notice that the estimate in the "parent" cell is *never* more than the estimate in the "child" cell? Can you figure out why this is?



A "child" cell could never have a lower estimate than a "parent" cell. If it did, this would mean that the "parent" cell's estimate was *not* the lowest possible; we would have just found a way for it to be *lower*, which is contradictory. So when a "parent" cell puts options to get to "child" cells onto the priority queue, the estimates for these options are *never less than the estimate for the parent*. If the parent's estimate is x, then it will only put options with estimates of x or higher onto the priority queue.

The key implication of this is that if there is a way to get to a cell that assigns it an estimate of x, then we know that we won't inadvertently accept a higher estimate, only to later discard the x estimate like our first attempt at this algorithm might have done. Estimates of x can only be put onto the queue by cells with an estimate of x or lower. So by definition, we know that once we're taking off options with an estimate of x + 1, we have already exhausted all possible ways of getting estimates of x. Each estimate value "tier" has all of its members identified before and during the processing of that "tier".

No cell can have an estimate *lower* than that of the top-left cell.

Because of all this, the order that A\* visits cells is in order of the best possible estimate that could be assigned to them. The estimate assigned to the top-left cell is used as the starting point; the algorithm then identifies all cells that share this same estimate. If it finds the bottom-right cell now, then it knows it couldn't have possibly done better than this. If it doesn't, though, it then essentially adds 1 to that estimate and identifies all cells that can be assigned that new estimate. And so forth, until it finds the bottom-right cell, which it can guarantee was reached with the smallest possible path length.

This is somewhat similar to BFS; remember that BFS assigns distances to cells that represent the shortest path from the top-left corner to that cell. Cells are explored in non-decreasing order of these distances. For A\*, we are assigning the most optimistic possible estimate to each cell and then exploring cells in non-decreasing order of these estimates.

*Proof of Correctness*

Note that this section is fairly advanced and is an optional extra aimed at those who are confident with college-level mathematics. It pulls together the intuition we investigated in the previous section.

The key idea is that the estimates for any given path from the top-left to bottom-right cell are *non-decreasing*; recall that the estimates are for the *shortest possible path* from the top-left to the bottom-right if we were to go through that cell. In A\* terminology, we say the estimate heuristic is *admissible* for having this property.

How do we know A\* will find the correct solution? Recall that if we have finished the algorithm, this means we have assigned some estimate, x, to the bottom-right cell. This estimate also happens to be the length of the shortest path to the bottom-right cell (the "remaining distance" component of the estimate is 0). So, how do we know there is not a path that would give an estimate b where b < x?

We can prove that such a b could not exist by using contradiction.

Consider the assumption that a path with an estimate of b for the bottom-right cell exists. If this is the case, we can write the path that leads to this better estimate as a list of nodes best\_path = [start, node1, node2,…, goal]. Remember that along this path, the estimate is non-decreasing. This means that every node in this path must have an estimate of less than x. We also know that this path shares at least one node with the path that leads to an estimate of x (at the very least, the start node was shared).

Now, remember that when we explore a node, we add its neighbors to be explored later. This means at least one of the nodes, node, in best\_path is still on the priority queue and unexplored. But this is a contradiction, as we know node has to have an estimate *less* than x, and so should have been explored first.

From this, we can conclude A\* will definitely find the shortest path.

**Algorithm**

We'll keep track of potential options into cells that we haven't yet visited using a **priority queue**. To keep track of distances, we'll simply use the "distances on queue" method from Approach 2. This means that we'll be putting 4 values onto the queue for each option; row, column, distance so far, and estimate of the total distance.

Because we might find more than one option for each cell, we need to check whether or not the cell has already been visited within the outer loop.

total\_rows = number of rows in grid

total\_cols = number of cols in gird

visited = a new set

queue = a new PRIORITY queue

enqueue (0, 0, 1, max of total\_rows and total\_cols)

add (0, 0) to visited

while the queue is not empty:

row, col, distance, estimate = dequeue and unpack an option

if (row, col) is the bottom right cell:

return distance

if (row, col) in visited:

skip processing this option

add (row, col) to visited

for each unvisited neighbor of (row, col):

otherwise calculate best case estimate to end

enqueue option

(neighbor\_row, neighbor\_col, distance + 1, estimate + distance + 1)

return -1

Notice that we do **not** add *neighbor* to visited. Doing so would be assuming that the first way into neighbor that was found is the best way; we saw above that this is not necessarily true.

The best-case estimate to the end is simply the maximum out of the number of rows and columns left to traverse.

**Code**

class Solution:  
 def shortestPathBinaryMatrix(self, grid: List[List[int]]) -> int:  
  
 max\_row = len(grid) - 1  
 max\_col = len(grid[0]) - 1  
 directions = [  
 (-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]  
  
 # Helper function to find the neighbors of a given cell.  
 def get\_neighbours(row, col):  
 for row\_difference, col\_difference in directions:  
 new\_row = row + row\_difference  
 new\_col = col + col\_difference  
 if not (0 <= new\_row <= max\_row and 0 <= new\_col <= max\_col):  
 continue  
 if grid[new\_row][new\_col] != 0:  
 continue  
 yield (new\_row, new\_col)  
  
 # Helper function for the A\* heuristic.  
 def best\_case\_estimate(row, col):  
 return max(max\_row - row, max\_col - col)  
  
 # Check that the first and last cells are open.   
 if grid[0][0] or grid[max\_row][max\_col]:  
 return -1  
  
 # Set up the A\* search.  
 visited = set()  
 # Entries on the priority queue are of the form  
 # (total distance estimate, distance so far, (cell row, cell col))  
 priority\_queue = [(1 + best\_case\_estimate(0, 0), 1, (0, 0))]  
 while priority\_queue:  
 estimate, distance, cell = heapq.heappop(priority\_queue)  
 if cell in visited:  
 continue  
 if cell == (max\_row, max\_col):  
 return distance  
 visited.add(cell)  
 for neighbour in get\_neighbours(\*cell):  
 # The check here isn't necessary for correctness, but it  
 # leads to a substantial performance gain.  
 if neighbour in visited:  
 continue  
 estimate = best\_case\_estimate(\*neighbour) + distance + 1  
 entry = (estimate, distance + 1, neighbour)  
 heapq.heappush(priority\_queue, entry)  
  
 # There was no path.  
 return -1

**Complexity Analysis**

Let N be the number of cells in the grid.

* Time complexity : O(N \log N)*O*(*N*log*N*).

The difference between this approach and the previous one is that adding and removing items from a priority queue is O(\log N)*O*(log*N*), as opposed to O(1)*O*(1). Given that we are adding/ removing O(N)*O*(*N*) items, this gives a time complexity of O(N \log N)*O*(*N*log*N*).

* Space complexity : O(N)*O*(*N*).

Same as the previous approaches.

Interestingly, there are ways to reduce the time complexity back down to O(N)*O*(*N*). The simplest is to recognize that there will be at most 3 unique estimates on the priority queue at any one time, and so to maintain 3 lists instead of a priority queue. Adding and removing from lists is O(1)*O*(1), bringing the total time complexity back down to O(N)*O*(*N*).