**Pick Your Battles**

Marijn Verschuren  
PCN: 510936

# 1. Table of contents

[1. Table of contents 2](#_Toc1)

[2. Algorithms 3](#_Toc2)

[2.1. The way out 3](#_Toc3)

[2.2. Send more money! 3](#_Toc4)

[2.3. From X to Y 3](#_Toc5)

[2.4. Shortest paths 3](#_Toc6)

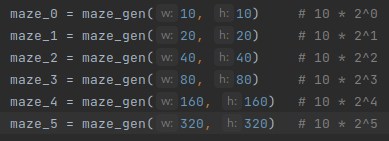
[2.5. Longest Common Subsequence 3](#_Toc7)

# 2. Algorithms

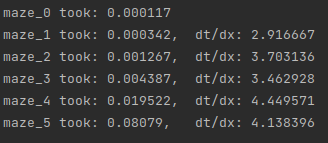
## 2.1. The way out

For this challenge I would use the A\* algorithm because it is one of the best path finding algorithms. A\* is a greedy algorithm with complexity of: O(E) in our case where E is the amount of edges. I have solved this in a project found [here](https://github.com/MarijnVerschuren/Tech/blob/main/s4/algo/PYB/main.py).

I did run into some issues with the maze generation because they are basically random noise, next time i will make my own implementation instead of modifying an existing one.



Test cases that result in the following data:



You can see that doubling the size of the graph will in (general) increase the algorithms time by a factor of 4 which is what you would expect given the previous analysis which yielded: O(E) where E is the amount of edges which scales with a factor of size^2 (actual: 2 \* (size \* size - 1)) due to the nature of the edges in a 2D grid.

## 2.2. Send more money!

For this challenge first check the length of string3 to be the same length ore one longer than string1,2. Then i would use an array of nested loops to find the values for each letter so that the equation is satisfied this will be a backtracking algorithm. This is admittedly a VERY slow process boasting a big oh of: O(10^n) where n is the amount of unique letters in the equation.

## 2.3. From X to Y

First calculate y/x and set m to it then use rule one: X := X \* m which will result in the following equation: X \* (Y/X) which will simplify to just: Y. This process will result in a big oh of: O(1)

## 2.4. Shortest paths

I would use Dijkstra’s algorithm (a greedy algorithm) to find the shortest paths between all nodes. The complexity is O(E \* logV) since that is the complexity of Dijkstra’s algorithm.

## 2.5. Longest Common Sub-Sequence

## I would use an algorithm that recursively loop from the end from the strings and checks for matches. Python code:

|  |
| --- |
| def GetLCS(str1, str2):  def recursive\_lcs(i, j):  if i == 0 or j == 0: return ''    if str1[i - 1] == str2[j - 1]:  return recursive\_lcs(i - 1, j - 1) + str1[i - 1]    lcs1 = recursive\_lcs(i, j - 1)  lcs2 = recursive\_lcs(i - 1, j)    return lcs1 if len(lcs1) > len(lcs2) else lcs2  return recursive\_lcs(len(str1), len(str2))  str1 = 'shdfjkbfruihejf'  str2 = 'ghitvgfgfuihejf'  result = GetLCS(str1, str2)  print(result) |

## The algorithm will check if the current chars match up if so the LCS of the rest of the strings is calculated. Otherwise the LCS’s of string A and string B - 1 char and vice versa are calculated

## The algorithm is classified as a backtracking algorithm with a complexity of: O(NM) where N is the length of str1 and M is the length of str2.