Sudoku problem:
All algorithms are implemented in sodokuSolver.py file. The sodokuSolver.py requires at lease one argument.
And a more file named sudokuTest.py is to test the ten times of the function in sodukuSolver, counting its time or even change the argument the number of given digits in the puzzle.

sodokuSolver.py argument

When argument is 'backtracking', the methods would call backtracking as the solver.
When argument is 'fancy', the methods would call the minimum remaining values (MRV) heuristic + forward checking + arc consistency as the solver.

Both question would use: mark_unavailable and mark_available in sudokuSolver.py
The mark_unavailable function is to add constrain for the given point's 'neighbors'.
The mark_available function is to reduce constrain for the given point's 'neighbors'.
* Neighbors means those points in the same raw, same column, and same 3*3 array with the given point.

Question 4:
Backtracking algorithm.
The 1st random Sudoku using backtracking with time: 0.096073 seconds.
The 2nd random Sudoku using backtracking with time: 0.003040 seconds.
The 3rd random Sudoku using backtracking with time: 0.003310 seconds.
The 4th random Sudoku using backtracking with time: 0.007054 seconds.
The 5th random Sudoku using backtracking with time: 0.102896 seconds.
The 6th random Sudoku using backtracking with time: 0.233245 seconds.
The 7th random Sudoku using backtracking with time: 0.074920 seconds.
The 8th random Sudoku using backtracking with time: 0.005246 seconds.
The 9th random Sudoku using backtracking with time: 0.026605 seconds.
The 10th random Sudoku using backtracking with time: 0.032366 seconds.

The average performance is: 0.05847
The variance of the performance is: 0.004749

Evaluation:
1. The backtracking algorithm works well when there are more given digits at beginning of the puzzle. If there are few given digits at start, leaving lots of blank to be filled, there would be lots of backtracking work costing lots of time.
2. The backtracking is unstable, which means the average variance is big due to the different random puzzle. The performance of this algorithm greatly depends on the puzzle itself. The run time of each puzzle would vary from tenth to hundredth.

The main function of this algorithm is solve_puzzle_backtracking in sudokuSolver.py.
Using depth first recursion to implement the backtracking.
canPut function is for checking the valid value to the points.

Question 5:
Fancy algorithm with minimum remaining values (MRV) heuristic + forward checking + arc consistency to the algorithm.
The 1st random Sudoku using fancy with time: 0.002074 seconds.
The 2nd random Sudoku using fancy with time: 0.002006 seconds.
The 3rd random Sudoku using fancy with time: 0.002536 seconds.
The 4th random Sudoku using fancy with time: 0.002815 seconds.
The 5th random Sudoku using fancy with time: 0.001998 seconds.
The 6th random Sudoku using fancy with time: 0.002240 seconds.
The 7th random Sudoku using fancy with time: 0.002035 seconds.
The 8th random Sudoku using fancy with time: 0.001917 seconds.
The 9th random Sudoku using fancy with time: 0.005555 seconds.
The 10th random Sudoku using fancy with time: 0.001946 seconds.

The average performance is: 0.002512
The variance of the performance is: 0.000001103

Evaluation:
1. The fancy algorithm wouldn't have different performance due to the position of the blank value. The fancy algorithm also has some backtracking designs.
2. The fancy algorithm is optimal algorithm and reduces many redundancies steps.
3. Compared with the backtracking algorithm, the fancy algorithm is much more stable. It wouldn't change its performance greatly for different puzzles.

The main function is solve_puzzle_fancy. We use a cube (3D array) to store each point's value and their relations' value.

Cube [x][y][value]

(x , y) means the position of a point.

When relation's value is 0, it stores the number of available values in this position. When relation's value is from 1 to 9, it stores the number of constrains. For example, if the position(x,y), and in the same row has a '3', and the same colum has a '3'. So Cube[x][y][3] = 2, means it has 2 constrains. If the row's 3 change to another number, the Cube[x][y][3] then reduce to 1. When the constrain reduce to 0, means we could add the number to this position.

Init_forward_checking is initiate the cube.

find_next_block function is to find the point where has the minimum available numbers.
check_arc function is to find if its ‘neighbors’(same column, same raw, in same 3*3 array) available number reduce to 0 after add certain number to this point.
solve_puzzle_fancy is the main fancy function.