Student Name: Mert Ekici
Student ID: 26772

Algorithms (CS301)
Assignment 4

Sabancı Üniversitesi

Agricultural Robotics Problem

Consider a farm of a rectangular shape, that is divided into n×msquare blocks. Some of these blocks contain weeds (i.e., unwanted plants that grow in-between crops). For instance, the left figure below illustrates a farm of size 5 × 6, where the blocks that contain weeds are denoted by black circles.

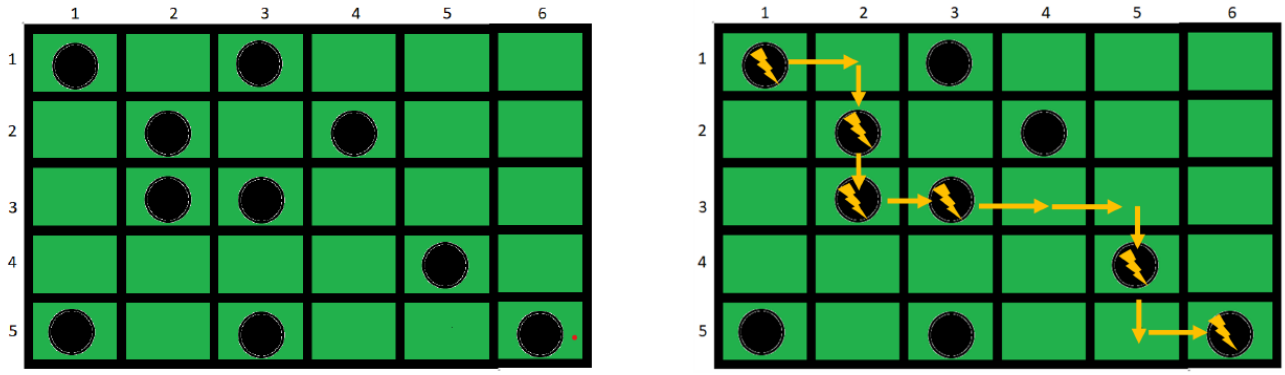

Figure 1: Field Map

Consider an agricultural mobile robot that is initially located at block (1, 1). The robot can navigate one block at a time, either to the right or to down. When the robot is at a block, it removes all the weeds in it. Given the information about which blocks contain weeds, the goal is for the robot to remove weeds from as many blocks as possible, and reach the charging area located at block (n,m). For instance, for the farm shown above, the robot can remove weeds from 6 blocks as illustrated in the right figure. Describe a dynamic programming algorithm to find a path that the robot can follow to remove weeds from the maximum number of blocks.

(a) Recursive formulation of this agricultural robotics problem

Recursive Formulation
given m[i,j] (matrix of the field -> map)
$i = [1, n]$ and $j = [1, m]$

$$f(i,j) = \begin{cases} m[i,j], & \text{if } i = j = 1 \\ f[i-1,j] + m[i,j] & \text{if } j = 1, \quad 1 < i \leq n \\ f[i,j-1] + m[i,j] & \text{if } i = 1, \quad 1 < j \leq m \\ max(f[i-1,j], f[i,j-1]) + m[i,j] & \text{otherwise} \end{cases}$$

To briefly explain this recursion, it calls the function f recursively based on the conditions, given the field map matrix. As a naive approach, it starts from the endpoint (5,6) coordinate in our case, and finds the path that removes the maximum amount of weed until the starting point. If the base condition is reached (i=j=1) it directly returns the value of this location which is the base condition. If i or j is equal to 1, it means that we can not move left or up and the function is called recursively only for the possible direction. Finally, if both ways are available (i and j are not equal to 1) then the maximum value returned for the recursion of either up or left direction is summed with the value of the current location (1 or 0).

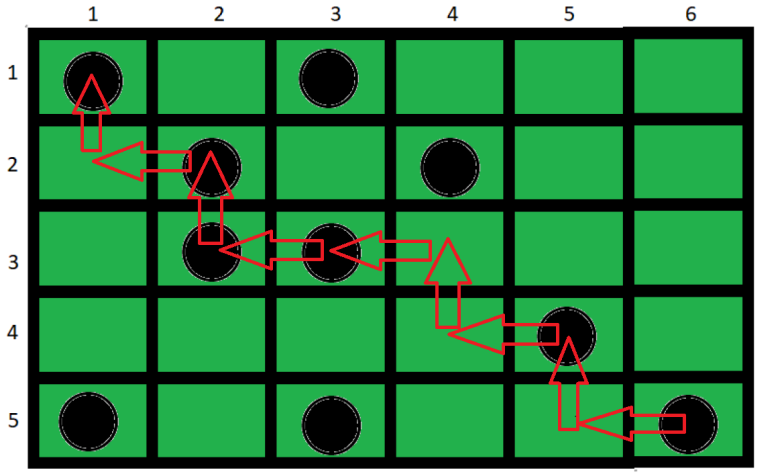

Figure 2: Recursive Solution

(b) Pseudocode of your algorithm designed using dynamic programming based on the recursive formulation

Based on the recursive formulation above, the following memoization algorithm is developed using dynamic programming to solve overlapping subproblems situation. Normally, naive recursive approach calls all possible paths to find the one with maximum number of weeds removed. However, this method is computationally inefficient. Since we call the function twice for each block (right or down) the asymptotic time complexity of the naive solution is $O(2^{n+m})$. In the Dynamic Programming solution based on memoization, we store the results of each subproblem when we first solve them. Then, if the result of a subproblem is already in the list (previously solved), we directly use it without computing again. This strategy lets us avoid unnecessary computational costs. Maximum Path Algorithm provided here consists of two functions: the inner utility function (recursive) *removeWeed()* and the main dynamic programming function *removeWeedDP()*. In the inner function, there are three cases: base condition where i or j are equal to 0 (edges of the table T) returns 0, else if the current location on the table is filled or this subproblem is previously solved then take it from the table and lastly, if the current location on the table is empty (problem is not solved yet) assign solution to the table by adding the maximum valued recursive calls on the top and left moves with the current location's value. Main DP function initializes n and m values, calls the inner utility function starting from the end point (recursion goes until the start point in a top-down manner), uses table to find the path with maximum value (reconstructs the path by tracing backwards in a bottom-up way) and returns the reversed path.

Algorithm 1 Maximum Path Algorithm (DP with Memoization)

Input: Field matrix $\boldsymbol{F}$, current row index $\boldsymbol{i}$, current column index $\boldsymbol{j}$, table for memoization $\boldsymbol{T}$
Output: ***count*** maximum number of weed blocks removed

function REMOVEWEED(F,i,j,T)
  if $i = 0$ or $j = 0$ **then**
    return 0
  else if $T[i][j] \neq -1$ **then**
    return T[i][j]
  else
    T[i][j] = max(removeWeed(F,i-1,j,T) , removeWeed(F,i,j-1,T)) + F[i-1][j-1]
    return T[i][j]
  end if

Input: Field matrix $\boldsymbol{F}$
Output: $\boldsymbol{P}$ path with maximum number of weed blocks removed

function REMOVEWEEDDP(F)
  Initialize n = number of rows in F and m = number of columns in F
  Initialize the memoization table T of size n+1 rows and m+1 columns filled with -1 values
  REMOVEWEED(F,m,n,T)
  Initialize the path P
  Add (n-1,m-1) to P which is the end point
  while $n > 1$ or $m > 1$ **do**
    if T[n-1][m]>T[n][m-1] **then**
      n=n-1
    else
      m=m-1
    end if
    Add (n,m) to P
  end while
  Reverse the path P
  return P

(c) Asymptotic time and space complexity analysis of your algorithm

After computing a solution to a subproblem, we now store it in a table. Subsequent calls check the table to avoid recomputation of the same results over and over again. Memoization as a top-down approach starts from the large input size (endpoint), reaches the smallest version of the problem (base case - starting point), and while doing this stores subproblem solutions to avoid overlapping. Since there is a constant work per table entry in memoization asymptotic time complexity of our algorithm is $O(m*n)$ where m is the number of rows and n is the number of columns. Similarly, all of these entries stored in the memory which makes the space complexity $O(m*n)$ as well. Let's analyze and derive these complexities from the steps of the algorithms.

1. All the initializations are constant time operations and executed once, $O(1)$ for both.
2. In the RemoveWeed function, assignment operations are done just for the first time when the subproblem is encountered. If the same problem appears again, its value is obtained from the table. Therefore, there is $O(m \times n)$ work (time complexity) and $O(m \times n)$ space required from the memory (space complexity).
3. Then, RemoveWeedDP function extracts the path from the table which the inner function returned by iterating backward on maximum values from the endpoint and adding these locations to the path. This while loop iterates for $m+n$ times which is the only possible path length to reach the endpoint from the start point. Hence time and space complexity of this part are $O(m+n)$.
4. Finally path is reversed which has O(m+n) time complexity (all items are iterated and swapped with the opposite one) and O(1) space complexity since no auxiliary space is used.

$$\textbf{Time Complexity: } O(1) + O(m*n) + O(m+n) + O(m+n) = O(m*n)$$
$$\textbf{Space Complexity: } O(1) + O(m*n) + O(m+n) + O(1) = O(m*n)$$

```python
def removeWeed(f,i,j,table):
    if (i==0 or j==0):
        return 0
    elif (table[i][j] != -1):
        return table[i][j]
    table[i][j] = max(removeWeed(f,i,j-1,table),removeWeed(f,i-1,j,table)) + f[i-1][j-1]
    return table[i][j]


def removeWeedDP(f):
    n,m = len(f), len(f[0])
    table = [[-1]*(m+1) for c in range(n+1)]
    wNum = removeWeed(f,n,m,table)
    path = []
    if(n>0 and m>0):
        path.append([n-1,m-1])
    while (n > 1) or (m > 1):
        if(table[n-1][m]>table[n][m-1]):
            n=n-1
        else:
            m=m-1
        path.append([n-1,m-1])
    path.reverse()
    return wNum, path, table
```

(d) Experimental evaluations of your algorithm: plot the results in a graph, and discuss the results

In this part, we will evaluate our algorithm experimentally. Then, results will be demonstrated and discussed. During the experimental evaluations, two testing methods are followed *Functional Testing* for the correctness of the algorithm and *Performance Testing* for evaluating the performance of the algorithm in practice. Before moving these tests, we need to implement random map generator.

Random Map Generator

```python
import random

# n is row size and m is column size, if not given they are selected randomly
def randMap(n=random.randint(0,100),m=random.randint(0,100)):
    farm = []
    for i in range(n):
        row = []
        for j in range(m):
            row.append(random.randint(0,1))
        farm.append(row)
    return farm
```

This function generated random field matrix to test the algorithm. If n and m parameters are not entered during the function call, they are randomly assigned by default. Given n and m, it created random field matrix sized $n \times m$ filled with 1s and 0s.

Example output of the randMap() function

$$\begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

Functional Testing

To test the correctness of the algorithm, black-box and white-box testing approaches will be utilized under this section.

Black-box testing is based on requirements and does not need source code and internal data. Under this method, extremes and edge case should be tested which are selected as:

- empty map
- one block
- all of the empty
- all of them filled
- linear (1 row, n columns)
- random

```
***Test#1***
map:
[]
max blocks removed: 0
path with max removal[]

***Test#2***
map:
[1]
max blocks removed: 1
path with max removal[[0, 0]]

***Test#3***
map:
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
max blocks removed: 0
path with max removal[[0, 0], [1, 0], [2, 0], [3, 0], [4, 0], [4, 1], [4, 2], [4, 3], [4, 4], [4, 5]]

***Test#4***
map:
[1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1]
max blocks removed: 10
path with max removal[[0, 0], [1, 0], [2, 0], [3, 0], [4, 0], [4, 1], [4, 2], [4, 3], [4, 4], [4, 5]]

***Test#5***
map:
[1, 0, 1, 1, 1, 0, 1, 0]
max blocks removed: 5
path with max removal[[0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [0, 5], [0, 6], [0, 7]]

***Test#6***
map:
[1, 0, 1, 1, 1]
[1, 1, 0, 1, 0]
[0, 1, 1, 0, 1]
max blocks removed: 6
path with max removal[[0, 0], [1, 0], [1, 1], [2, 1], [2, 2], [2, 3], [2, 4]]
```

Figure 3: Output of Black-Box Test Suite (txt)

With the given cases, implemented removeWeedDP function correctly solves each maximum sum path problem by returning correct maximum block removal value with the correct path.

In the **White-box testing**, testers have access to the system design. They can examine the documents and view the code. There are several coverage criteria that can be used in this testing method:

- Statement coverage (try test cases that will exercise every statement)
- Decision coverage (try test cases that will exercise every decision)
- Path coverage (try test cases that will exercise every path)

The test suite created for Black Box testing is analyzed based on statement coverage in here, White Box testing.

1. BBf1 (empty map) enters if in removeWeed, passes if and while parts in removeWeedDP.
2. BBf2 (one block) enters both if and elif conditions in removeWeed, enters if but passes while in removeWeedDP.
3. BBf3 (all empty) covers all statements.
4. BBf4 (all filled) covers all statements.
5. BBf5 (linear) enters both if and elif conditions in removeWeed, enters the outside if but in while it only enters the else part in removeWeedDP.
6. BBf6 (random) covers all statements.

This test suite has 100% Statement Coverage.

Performance Testing

Performance testing measures the running time of the program by trying various/growing input sizes. As discussed in part c, asymptotic time complexity of the algorithm is $O(m * n)$ which is linear with the matrix size. To test the input size's effect on the algorithms running time, square matrices with $n \times n$ size are used. They all randomly generated in a loop (by randMap() function) and running times are measured for the sizes between 1×1 to 400×400. The loop increments n values one-by-one until it reaches 400. Thus, the size increases as the square of n values. Figure 4 and 5 below show the increasing running time in seconds with the growing matrix size. As expected, execution time grows linearly with the input size such that time complexity of the algorithm is $O(m * n)$. When the plot is modified into a version where x values are n values (square roots of the input sizes), now elapsed time grows quadratic (polynomial) since the size increases as the square of n values and running time has the same behavior as well (because it is linear with the size).

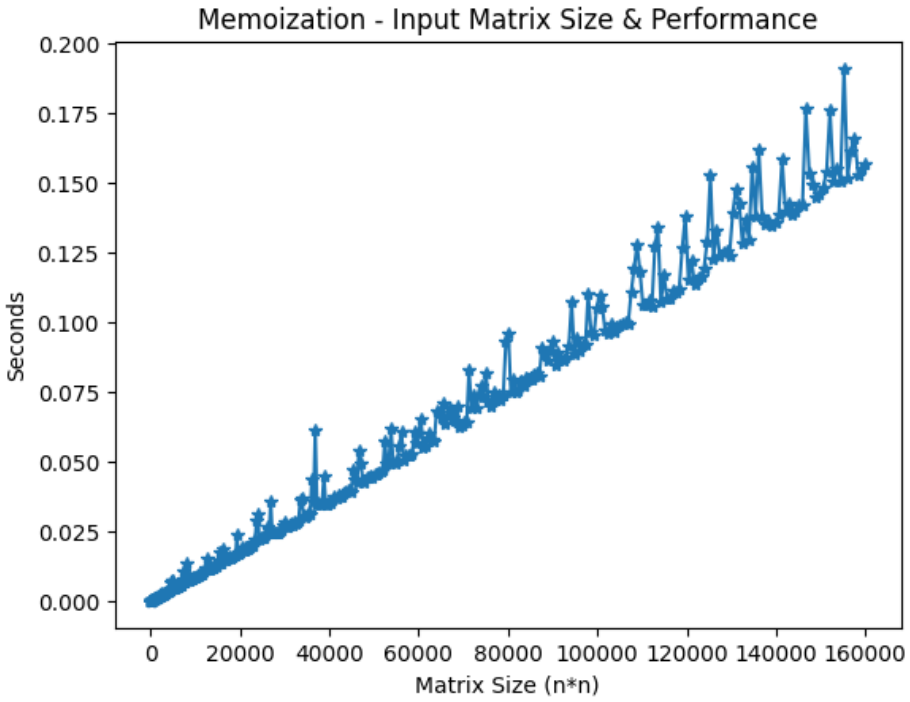

Figure 4: Memoization - Input Matrix Size and Performance

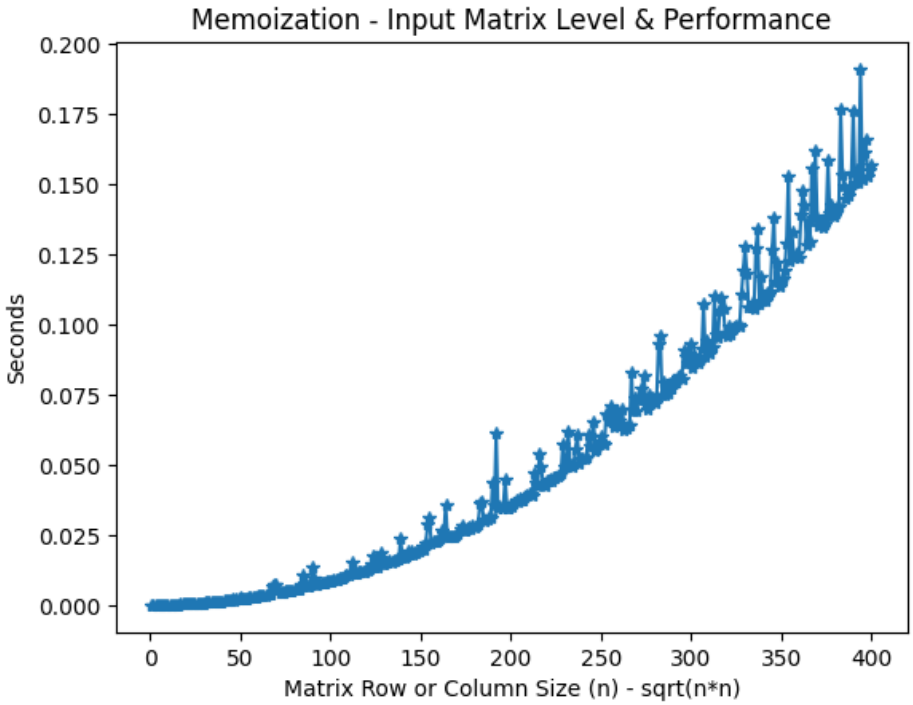

Figure 5: Memoization - Input Matrix Level and Performance

Finally, Central Limit Theorem is used to obtain reliable runtime information about our algorithm. Single measurement is always a risky choice where the measured values can easily fluctuate and get affected by instant conditions. Given a sufficiently large sample size from a population with finite variance, the mean of all samples from the same population approximately equal to the mean of the real population. Following image shows the mean elapsed time values for N=20 sample sized executions and compares it with the single measurements.

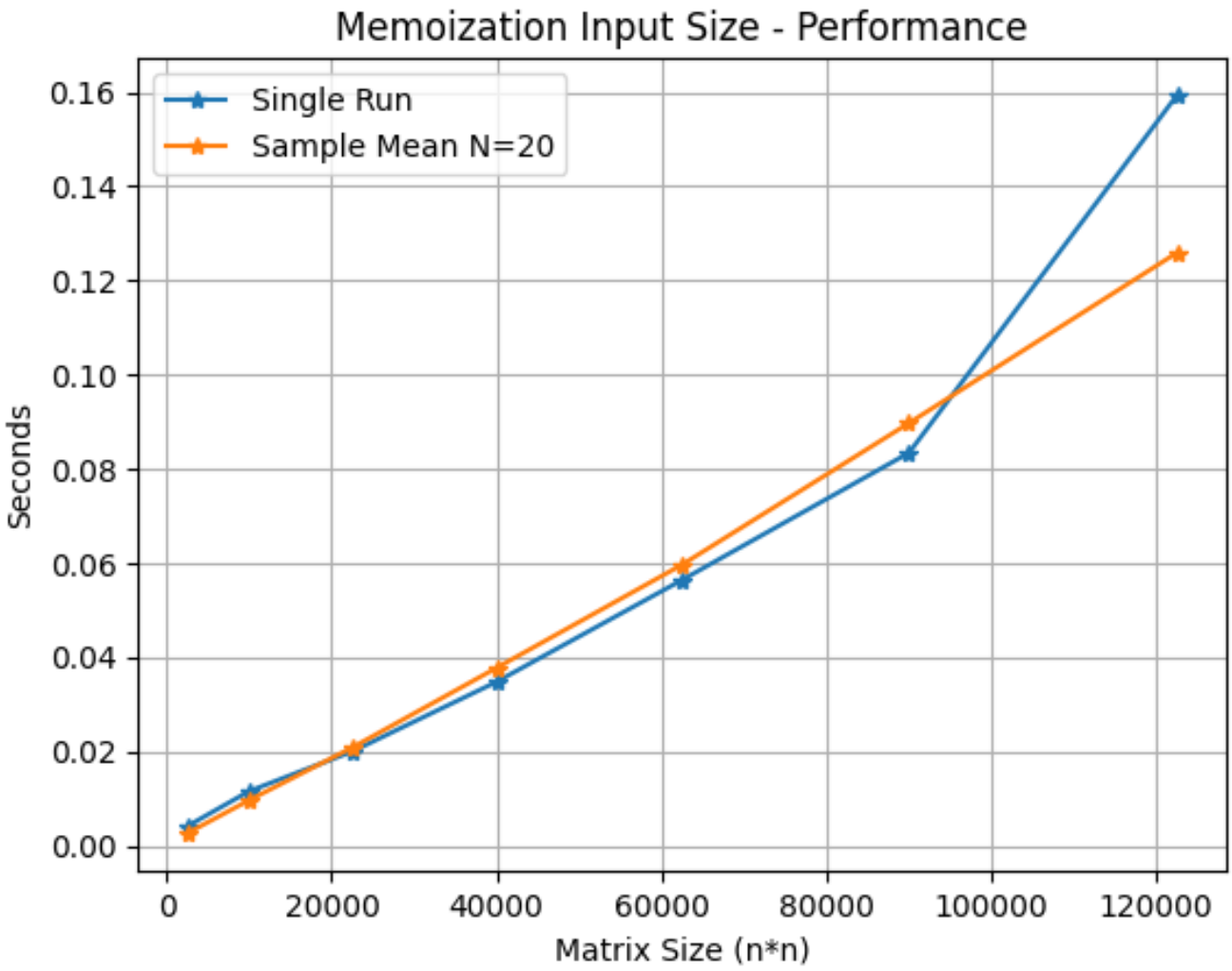

Figure 6: Memoization - Input Matrix Level and Performance

For n values 50, 100, 150, 200, 250, 300 and 350 (n×n), CLT based mean values seem more consistent and linear compared to the single run results.

In the last part, 90% and 95% confidence intervals are calculated for each n value tested for N=20 sized samples using the Central Limit Theorem's $m \pm t \times s_m$ (Figure 7).

```
Confidence interval of average run time for 50 x 50 matrix (90% confidence interval): 0.005020798098118272 - 0.00020643810189843303
Confidence interval of average run time for 50 x 50 matrix (95% confidence interval): 0.005527572834562466 - -0.0003003366345457601

Confidence interval of average run time for 100 x 100 matrix (90% confidence interval): 0.013580019434028187 - 0.005675508665978625
Confidence interval of average run time for 100 x 100 matrix (95% confidence interval): 0.014412073199086037 - 0.004843454900920776

Confidence interval of average run time for 150 x 150 matrix (90% confidence interval): 0.024302631708626423 - 0.01730442669139168
Confidence interval of average run time for 150 x 150 matrix (95% confidence interval): 0.025039284868335343 - 0.01656777353168276

Confidence interval of average run time for 200 x 200 matrix (90% confidence interval): 0.043098524269090915 - 0.03236999663088082
Confidence interval of average run time for 200 x 200 matrix (95% confidence interval): 0.04422784296784987 - 0.031240677932121864

Confidence interval of average run time for 250 x 250 matrix (90% confidence interval): 0.06419824053276392 - 0.05511599796725756
Confidence interval of average run time for 250 x 250 matrix (95% confidence interval): 0.0651542660659751 - 0.05415997243404636

Confidence interval of average run time for 300 x 300 matrix (90% confidence interval): 0.09719142173822251 - 0.08225310796178134
Confidence interval of average run time for 300 x 300 matrix (95% confidence interval): 0.09876387581995316 - 0.08068065388005069

Confidence interval of average run time for 350 x 350 matrix (90% confidence interval): 0.13340286147943642 - 0.11841995452058986
Confidence interval of average run time for 350 x 350 matrix (95% confidence interval): 0.13498000958036763 - 0.11684280641965863
```

Figure 7: Confidence Intervals