Kaunas University of Technology
Faculty of Informatics

Recursion versus Dynamic Programming

Engineering Project Report Task Variant No. 12

ORKUN MANAP

1 Task No. 1

For the given recurrence formula

$$F(n) = \begin{cases} F(n-3) + 4F\left(\frac{n}{6}\right) + 9F\left(\frac{n}{7}\right) + \frac{n^3}{2}, if\ n > 1 \\ 1, \quad otherwise \end{cases}$$

design two algorithms and evaluate their complexity:

- directly applying recursion.
- using the property of dynamic programming – possibility to store partial solutions (memorization).

Implement in software these two algorithms and experimentally evaluate and compare the time complexity of both algorithms.

1.1 Formula calculation function using recursion and dynamic programming.

Recursion. Using the recursive solution, such a function can be constructed to calculate a given formula (Figure 1).

```
# DEFINE ALGORITHMS

def func_re(n):
    if n > 1:
        return func_re(n - 3) + 4 * func_re(n // 6) + 9 * func_re(n // 7) + (n ** 3) / 2
    else:
        return 1
```

Figure 1. Formula calculation using recursion.

As can be seen in formula, if n is bigger than 1, it will return recursion function. However, if it is lower than or equal to 1 it will return 1.

Memorization. Using dynamic programming algorithm, we will employ the property that it is possible to memorize (store) the results of partial solutions (Figure 2).

```
def func_re(n):
    if n > 1:
        return func_re(n - 3) + 4 * func_re(n // 6) + 9 * func_re(n // 7) + (n ** 3) / 2
    else:
        return 1

def memoize(func):
    cache = dict()

    def memoized_func(*args):
        if args in cache:
            return cache[args]
        result = func(*args)
        cache[args] = result
        return result

    return memoized_func

func_memo = memoize(func_re)
```

Figure 2. Formula calculation using memorization.

To store (cache) data, dictionary (which is variable in Python) and arguments are used. Memoize function [1]

1.2 Checking the correctness of the formula

Let us check that both calculation algorithms give the same results when *n* is small (Table 1). The text of the program is provided in Appendix 1.

Table 1. Comparison of formula values

n	F(N), Recursion	F(N), Memoization
1	1.00e+00	1.00e+00
2	1.80e+01	1.80e+01
3	2.75e+01	2.75e+01
4	4.60e+01	4.60e+01
5	9.35e+01	9.35e+01
6	1.48e+02	1.48e+02
7	2.30e+02	2.30e+02
8	3.62e+02	3.62e+02
9	5.26e+02	5.26e+02
10	7.44e+02	7.44e+02

As can be seen from the table 1, results of the recursion and memorization functions are same. That means, memorization function working without any problem.

1.3 Experimental evaluation of algorithm complexity

Recursion. Numerous experiments have been performed to determine what sample sizes are best suited to illustrate recursion runtime. Seven data samples were selected, n = 40, 80, …, 2560 (Table 2).

Table 2. Formula values and execution times using recursion. [2]

n	20	40	80	160	320	640	1280
F(n)	9.61e+03	1.30e+05	1.95e+06	3.11e+07	5.21e+08	9.37e+09	1.86e+11
Time	3.38e-05	6.82e-05	2.38e-04	1.24e-03	7.68e-03	5.32e-02	5.93e-01

As can have seen from the table, there is a direct proportion between number of variables and time as it expected.

The recursion time graph is presented in the figure window (Figure 3).

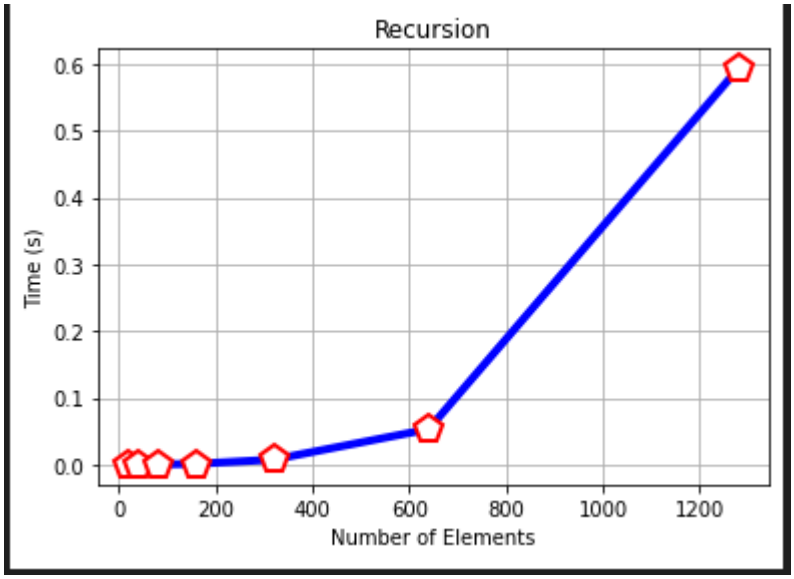


Figure 3. Recursion time chart

Memorization. Much larger samples had to be selected to calculate the dynamic programming (Table 3). Seven data samples were selected, n = 100, 200, …, 6400.

Table 3. Formula values and execution times using memorization.

n	100	200	400	800	1600	3200	6400
F(n)	4.73e+06	7.66e+07	1.31e+09	2.42e+10	5.03e+11	1.22e+13	3.61e+14
Time 1	1.14e-04	6.03e-04	3.71e-03	2.87e-02	3.15e-01	3.90e+00	6.08e+01
Time 2	1.10e-04	5.92e-04	3.67e-03	2.91e-02	3.14e-01	3.83e+00	6.03e+01
Time 3	1.10e-04	6.16e-04	3.79e-03	2.98e-02	3.15e-01	3.89e+00	6.13e+01
Time 4	1.11e-04	6.24e-04	3.89e-03	3.03e-02	3.25e-01	4.21e+00	6.33e+01
Time 5	1.17e-04	6.31e-04	4.08e-03	3.14e-02	3.47e-01	3.92e+00	6.18e+01

Time averages (excluding first experiment)						
Time	1.12e-04	6.16e-04	3.86e-03	3.02e-02	3.25e-01	3.96e+00	6.17e+01

Time averages were calculated minus times measured during the first experiment.

As can be seen in Table 3, there is a direct proportion between the number of variables and time. The function is quite fast with the memoisation technique. However, after 32000 values, even with the memoisation technique, it takes time to calculate it. But still, it is much more faster than recursive function without memoisation.

Averaged dynamic programming times are presented graphically (Figure 4).

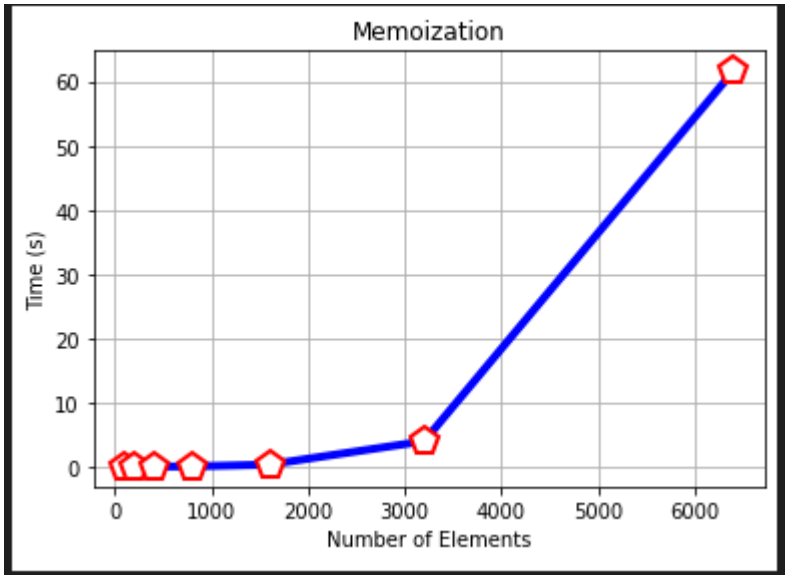


Figure 4. Dynamic programming time chart

What can we say about the time complexity of both methods (recursion and memoisation) by comparing their timetables?

When we compare Figure 3 and Figure 4. The curve is almost same. However, the number of elements is quite different. So, as expected, recursive function with memoisation is faster.

They both has similar curve with n^2 function. So, we can say that they have O(n) time complexity.

2 Task No. 2

Following the verbal description of a given task and using the dynamic programming method:

- design the algorithm of a task.
- implement the algorithm in software and experimentally evaluate algorithm's complexity.

The cells of the square board (n × n) are integers. The snake from the bottom right must face the top left. It can only creep into the box on the left or at the top.

You need to determine the route with the maximum sum of cells and calculate this sum.

Example:

A =

1 2 3 4

2 2 3 4

3 9 8 4

4 5 6 7

max_path_sum =

36

max_path =

7 6 8 9 3 2 1

The initial task disintegration into smaller subtasks (creating the recurrence relation)

For example, let's say we have array like this:

```
M = [[1, 2, 3, 4],
     [2, 2, 3, 4],
     [3, 9, 8, 4],
     [4, 5, 6, 7]]
```

We will start from last element which is 7. Because we can only move to up or left and we want to maximum cost, we will compare M [2][3] (which is 4) and M [3][2] (which is 6) at first. After that, we will go to 6 because 6 > 4.

```
M = [[1, 2, 3, 4],
     [2, 2, 3, 4],
     [3, 9, 8, 4],
     [4, 5, 6, 7]]
```

Then, we will compare M [2][2] (which is 8) and M [3][1] (which is 5).

As can be seen, we will not need those yellow parts anymore:

```
M = [[1, 2, 3, 4],
     [2, 2, 3, 4],
     [3, 9, 8, 4],
     [4, 5, 6, 7]]
```

Because we will not compare or use any of those number. For that reason, we can delete that column and row.

New sub matrix:

```
M = [[1, 2, 3],
     [2, 2, 3],
     [3, 9, 8]]
```

So, we do not have to use as much as resources anymore. And we can continue with this sub matrix.

One more step:

Compare -> 9>3, so we can remove this yellow part:

```
M = [[1, 2, 3],
     [2, 2, 3],
     [3, 9, 8]]
```

New sub matrix:

```
M = [[1, 2],
     [2, 2],
     [3, 9]]
```

2.1 Software realization of the task using dynamic programming.

Creating a dynamic programming algorithm, we will be using a property that we can memorize the values of partial solutions**.** A Python function that implements this dynamic programming feature and algorithm itself is provided (Figure 6).

```
def Snake(M, x, y, path, path_sum):
    global max_path
    global max_path_sum

    print('X: ' + str(x), 'Y: ' + str(y))
    print('M:', M)
    print("Value:", M[x][y])
    print("Path:", path)
    print("Path Sum:", path_sum)
    print('\n')

    max_path = max_path + [M[x][y]]
    max_path_sum = max_path_sum + M[x][y]

    if x > 0 and y > 0:
        if M[x-1][y] > M[x][y-1]:
            new_path = path + [M[x-1][y]]
            new_sum = M[x-1][y] + path_sum

            M = M[:][:x]

            Snake(M, x-1, y, new_path, new_sum)
        else:
            new_path = path + [M[x-1][y]]
            new_sum = M[x-1][y] + path_sum

            for i in M: del i[-1]

            Snake(M, x, y-1, new_path, new_sum)
    elif x > 0:
        new_path = path + [M[x-1][y]]
        new_sum = M[x-1][y] + path_sum

        M = M[:][:x]

        Snake(M, x-1, y, new_path, new_sum)
    elif y > 0:
        new_path = path + [M[x-1][y]]
        new_sum = M[x-1][y] + path_sum

        for i in M: del i[-1]

        Snake(M, x, y-1, new_path, new_sum)
```

Figure 6. Snake function

This part removes a row.

```
M = M[:][:x]
```

This part removes a column.

```
for i in M: del i[-1]
```

The testing program provided in Appendix 2 gives us the following results (Table 4).

Table 4. Steps of process

```
X: 3 Y: 3
M: [[1, 2, 3, 4], [2, 2, 3, 4], [3, 9, 8, 4], [4, 5, 6, 7]]
Value: 7
Path: []
Path Sum: 0


X: 3 Y: 2
M: [[1, 2, 3], [2, 2, 3], [3, 9, 8], [4, 5, 6]]
Value: 6
Path: [7]
Path Sum: 7


X: 2 Y: 2
M: [[1, 2, 3], [2, 2, 3], [3, 9, 8]]
Value: 8
Path: [7, 6]
Path Sum: 13
```

```
X: 2 Y: 1
M: [[1, 2], [2, 2], [3, 9]]
Value: 9
Path: [7, 6, 8]
Path Sum: 21


X: 2 Y: 0
M: [[1], [2], [3]]
Value: 3
Path: [7, 6, 8, 9]
Path Sum: 30


X: 1 Y: 0
M: [[1], [2]]
Value: 2
Path: [7, 6, 8, 9, 3]
Path Sum: 33
```

```
X: 0 Y: 0
M: [[1]]
Value: 1
Path: [7, 6, 8, 9, 3, 2]
Path Sum: 35
```

```
max_path
```

```
[7, 6, 8, 9, 3, 2, 1]
```

```
max_path_sum
```

```
36
```

X -> Current index of row.

Y -> Current index of column.

Value -> Current value.

Path -> Current path of snake.

Path Sum -> Current sum of path cost.

M -> Current matrix.

2.2 Experimental evaluation of algorithm complexity

Time measurement was performed by changing the number of elements. Time was recorded when the number of items n = [4, 16, 64, 256, 1024], i.e., the number of elements has been squared (Table 5).

Table 5. Snake timetable

n	4	16	64	256	1024
Time 1	5.39e-04	6.67e-03	4.45e-02	3.47e-01	8.00e+00
Time 2	1.48e-03	1.05e-03	2.67e-02	6.02e-01	7.73e+00
Time 3	1.62e-03	1.16e-03	2.92e-02	3.95e-01	7.12e+00
Time 4	1.38e-03	1.94e-03	1.85e-02	5.21e-01	8.90e+00
Time 5	1.38e-03	2.38e-03	2.30e-02	4.22e-01	9.00e+00

The experiment with 5 data samples was repeated five times. The first row and the first column of the table have been removed to eliminate the time loss due to compiling the loop for the first time.

The averages of the four experiments without the first sample were calculated:

Time averages					
Time	1.68e-03	1.96e-03	2.92e-02	5.25e-01	8.45e+00

These results are presented in a graphical window using a matplotlib.plot (Figure 7). [3]

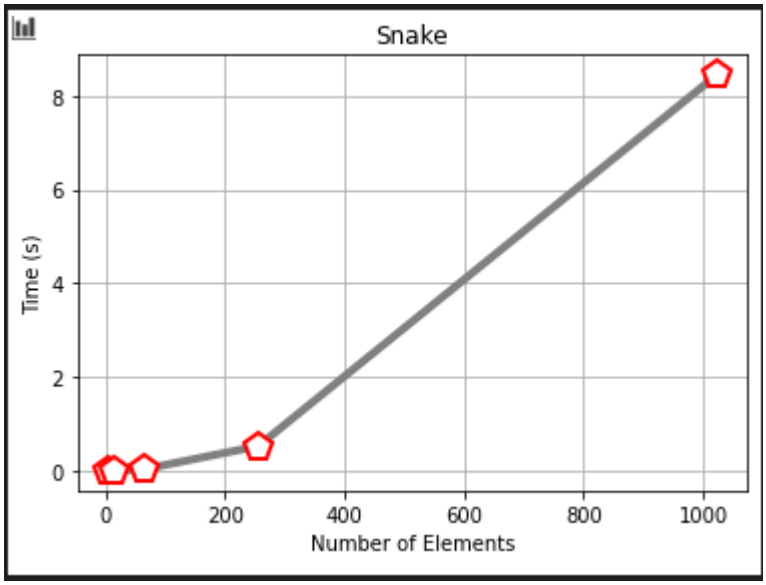


Figure 7. Snake time complexity

A linear approximation was performed for the chart points using the matplotlib library in Python. Taking into consideration the determination coefficient R^2 value, we can say that this algorithm has **almost** Linear Time Complexity $O(n)$.

Conclusions

A recursive function is a function that calls itself repeatedly while it is running and outputs an output each time it is called.

The recursive function allows programmers to write more efficiently using fewer lines of code. However, it has advantages as well as disadvantages. One of these is that it allows infinite loops if not properly encoded.

It is possible that we can make recursive functions even more efficient. One of these methods is to use the Memoization method. Memoization can be briefly explained as follows: A method of keeping previously calculated values in memory to avoid potential duplication. Since the calculated values are kept in memory, repeated calculations are avoided, which saves us time. However, no matter how fast this method is, one of the disadvantages of this method is that we have to figure it out manually how to split the main problem into sub-problems within the code. [4]

References

1. https://dbader.org/blog/python-memoization#:~:text=Memoization%20allows%20you%20to%20optimize,quickly%20retrieved%20from%20a%20cache.

2. https://pymotw.com/3/timeit/index.html

3. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html

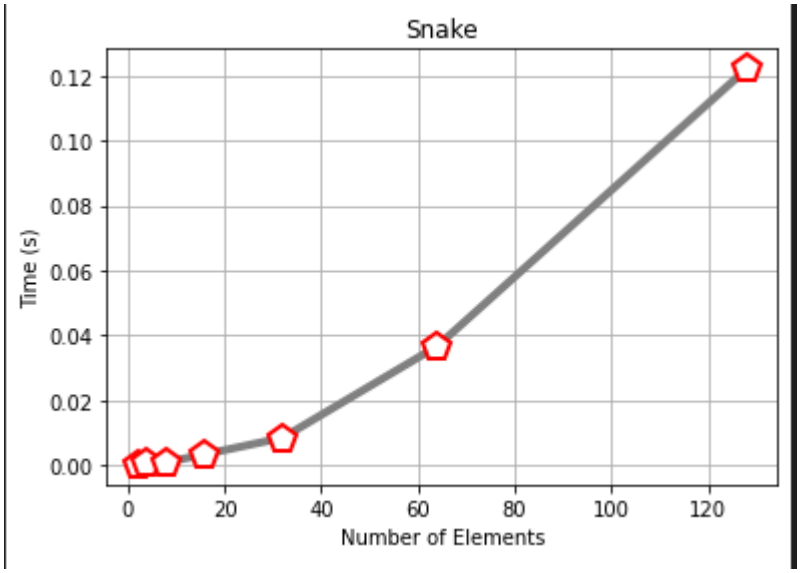
4. https://levelup.gitconnected.com/memoization-and-recursion-290f0e5a0351?gi=22566aa3ad87

3 After Defense

n	2	4	8	16	32	64	128
Time 1	1.93e-04	5.07e-04	8.70e-04	2.48e-03	1.20e-02	3.56e-02	1.15e-01
Time 2	1.46e-04	1.72e-04	9.23e-04	4.24e-03	6.79e-03	3.63e-02	1.03e-01
Time 3	1.50e-04	1.79e-04	5.03e-04	5.70e-03	9.13e-03	4.28e-02	1.31e-01
Time 4	1.99e-04	1.35e-03	7.77e-04	1.36e-03	7.24e-03	3.31e-02	1.32e-01
Time 5	1.42e-04	1.43e-04	1.94e-03	3.52e-03	9.90e-03	3.02e-02	1.29e-01
Time 6	1.66e-04	2.40e-04	3.08e-04	2.71e-03	9.12e-03	3.57e-02	1.35e-01
Time 7	1.44e-04	4.26e-04	9.10e-04	2.70e-03	8.31e-03	4.20e-02	1.03e-01

Time averages

Time	1.58e-04	4.18e-04	8.93e-04	3.37e-03	8.42e-03	3.67e-02	1.22e-01



Appendices

Appendix 1.

```
# IMPORTS

import functools
import timeit
import matplotlib.pyplot as plt
import sys
from datetime import datetime
```

```
# DEFINE VARIABLES

n = 11

n_values = [20, 40, 80, 160, 320, 640, 1280]

time_a = []
val_a = []
val_b = []
time_arr = []

a = []
b = []
```

```
# DEFINE ALGORITHMS

def func_re(n):
    if n > 1:
        return func_re(n - 3) + 4 * func_re(n // 6) + 9 * func_re(n // 7) + (n ** 3) / 2
    else:
        return 1

@functools.cache
def func_memo(n):
    if n > 1:
        return func_memo(n - 3) + 4 * func_memo(n // 6) + 9 * func_memo(n // 7) + (n ** 3) / 2
    else:
        return 1
```

```
def func_re(n):
    if n > 1:
        return func_re(n - 3) + 4 * func_re(n // 6) + 9 * func_re(n // 7) + (n ** 3) / 2
    else:
        return 1

def memoize(func):
    cache = dict()

    def memoized_func(*args):
        if args in cache:
            return cache[args]
        result = func(*args)
        cache[args] = result
        return result

    return memoized_func

func_memo = memoize(func_re)
```

```
# Checking the correctness of the formula

for i in range(1, n):
    a.append(func_re(i))

for i in range(1, n):
    b.append(func_memo(i))
```

```
print("----------------------------------------")
print(" n | F(N), Recursion | F(N), Memoization|")
for i in range(0, n - 1):
    print(str(i + 1) + " |      " + "{:.2e}".format(a[i]) + "    |      " + "{:.2e}".format(b[i]
) + "     |")

print("----------------------------------------")
```

```
# Recursion function time values

for i in n_values:
    start = timeit.default_timer()
    val_a.append(func_re(i))
    stop = timeit.default_timer()
    time_a.append(stop - start)
```

```
# Experimental evaluation of algorithm complexity

print
("-------------------------------------------------------------------------------------------")
print(" n   | 20        | 40        | 80        | 160        | 320        | 640       | 1280      |
")
print
("-------------------------------------------------------------------------------------------")
print("F(n) |", end="")

for i in range(7):
    print(" {:.2e}".format(val_a[i]), end="  |")

print("\nTime |", end="")
for i in range(7):
    print(" {:.2e}".format(time_a[i]), end="  |")

print
("\n-------------------------------------------------------------------------------------------")
```

```
# Plotting recursion function time values / number of elements
fig, ax = plt.subplots()
ax.plot(n_values, time_a, '-p', color='blue',
         markersize=15, linewidth=4,
         markerfacecolor='white',
         markeredgecolor='red',
         markeredgewidth=2)

ax.set(xlabel='Number of Elements', ylabel='Time (s)',
       title='Recursion')
ax.grid()
plt.show()
```

```
# Memoization function time values

n_values = [100, 200, 400, 800, 1600, 3200, 6400]

val_b = []
time_arr = []

sys.setrecursionlimit(2500)

for j in range(5):
    time_b = []

    def func_re(n):
        if n > 1:
            return func_re(n - 3) + 4 * func_re(n // 6) + 9 * func_re(n // 7) + (n ** 3) / 2
        else:
            return 1

    def memoize(func):
        cache = dict()

        def memoized_func(*args):
            if args in cache:
                return cache[args]
            result = func(*args)
            cache[args] = result
            return result

        return memoized_func

    func_memo = memoize(func_re)

    for i in n_values:
        print(i)
        start = timeit.default_timer()
        val_b.append(func_memo(i))
        stop = timeit.default_timer()
        time_b.append(stop - start)
    time_arr.append(time_b)
```

```
# Memoization

print
("-----------------------------------------------------------------------------------------------")
print(" n   | 100        | 200        | 400      | 800     | 1600       | 3200     | 6400     |
")
print
("-----------------------------------------------------------------------------------------------")
print("F(n)  |", end="")

for i in range(7):
    print(" {:.2e}".format(val_b[i]), end="  |")

for j in range(5):
    print("\nTime " + str(j + 1) + "|", end="")
    for i in range(7):
        print(" {:.2e}".format(time_arr[j][i]), end="  |")

print
("\n-----------------------------------------------------------------------------------------------")
```

```
# Calculation of average value of times

avg_arr = []
for i in range(7):
    avg_arr.append((time_arr[1][i] + time_arr[2][i] +
                    time_arr[3][i] + time_arr[4][i]) / 4)
```

```
# Table of avg

print("Time averages (excluding first experiment)")
print("\nTime |", end="")
for i in range(7):
    print(" {:.2e}".format(avg_arr[i]), end="  |")
```

```
# Plotting avg times / number of elements

fig, ax = plt.subplots()
ax.plot(n_values, avg_arr, '-p', color='blue',
         markersize=15, linewidth=4,
         markerfacecolor='white',
         markeredgecolor='red',
         markeredgewidth=2)

ax.set(xlabel='Number of Elements', ylabel='Time (s)',
       title='Memoization')
ax.grid()
plt.show()
```

Appendix 2.

```
import numpy as np
import timeit
import matplotlib.pyplot as plt
```

```
M = [[1, 2, 3, 4], [2, 2, 3, 4], [3, 9, 8, 4], [4, 5, 6, 7]]
```

```
starting_point_x, starting_point_y = 3, 3
max_path = []
max_path_sum = 0
```

```
def Snake(M, x, y, path, path_sum):
    global max_path
    global max_path_sum

    print('X: ' + str(x), 'Y: ' + str(y))
    print('M:', M)
    print("Value:", M[x][y])
    print("Path:", max_path)
    print("Path Sum:", max_path_sum)
    print('\n')

    max_path = max_path + [M[x][y]]
    max_path_sum = max_path_sum + M[x][y]

    if x > 0 and y > 0:
        if M[x-1][y] > M[x][y-1]:
            new_path = path + [M[x-1][y]]
            new_sum = M[x-1][y] + path_sum

            M = M[:][:x]

            Snake(M, x-1, y, new_path, new_sum)
        else:
            new_path = path + [M[x-1][y]]
            new_sum = M[x-1][y] + path_sum

            for i in M: del i[-1]

            Snake(M, x, y-1, new_path, new_sum)
    elif x > 0:
        new_path = path + [M[x-1][y]]
        new_sum = M[x-1][y] + path_sum

        M = M[:][:x]

        Snake(M, x-1, y, new_path, new_sum)
    elif y > 0:
        new_path = path + [M[x-1][y]]
        new_sum = M[x-1][y] + path_sum

        for i in M: del i[-1]

        Snake(M, x, y-1, new_path, new_sum)
```

```
Snake(M, starting_point_x, starting_point_y, [], 7)
```

```
max_path
```

```
[7, 6, 8, 9, 3, 2, 1]
```

```
max_path_sum
```

```
36
```

```
def Snake2(M, x, y, path, path_sum):
    global max_path
    global max_path_sum

    max_path = max_path + [M[x][y]]
    max_path_sum = max_path_sum + M[x][y]

    if x > 0 and y > 0:
        if M[x-1][y] > M[x][y-1]:
            new_path = path + [M[x-1][y]]
            new_sum = M[x-1][y] + path_sum

            M = M[:][:x]

            Snake2(M, x-1, y, new_path, new_sum)
        else:
            new_path = path + [M[x-1][y]]
            new_sum = M[x-1][y] + path_sum

            for i in M: np.delete(i,-1)

            Snake2(M, x, y-1, new_path, new_sum)
    elif x > 0:
        new_path = path + [M[x-1][y]]
        new_sum = M[x-1][y] + path_sum

        M = M[:][:x]

        Snake2(M, x-1, y, new_path, new_sum)
    elif y > 0:
        new_path = path + [M[x-1][y]]
        new_sum = M[x-1][y] + path_sum

        for i in M: np.delete(i,-1)

        Snake2(M, x, y-1, new_path, new_sum)
```

```
numbs = [4, 16, 64, 256, 1024]

max_path2 = []
max_path_sum2 = []

experiment_time = []
experiment_sum = []

for j in range(5):

    time_arr = []
    for i in numbs:

        max_path = []
        max_path_sum = 0

        print('-----------------------------------ITERATION FOR {0}.{1}
NUMBERS-------------------------------------'.format(j, i))

        start = timeit.default_timer()

        arr = np.random.uniform(low=0.5, high=10, size=(i, i))

        Snake2(arr, i-1, i-1, [], arr[i-1][i-1])

        max_path2.append(max_path)
        max_path_sum2.append(max_path_sum)

        stop = timeit.default_timer()

        time_arr.append(stop - start)

    experiment_time.append(time_arr)
    experiment_sum.append(max_path_sum)
```

```
# Calculation of average value of times

avg_arr = []
for i in range(5):
    avg_arr.append((experiment_time[1][i] + experiment_time[2][i] +
                    experiment_time[3][i] + experiment_time[4][i]) / 4)
```


```
# Calculation of average value of times

avg_arr = []
for i in range(5):
    avg_arr.append((experiment_time[1][i] + experiment_time[2][i] +
                    experiment_time[3][i] + experiment_time[4][i]) / 4)
```