TUMO Matrix Task

March 31, 2021

We are going to use Dynamic Programming. The idea for solving this is the following: We first build a table m, which is going to have length and height equal to the number of the elements. Then, we are going to take i, the rows, as a starting point, and j, the columns, as an ending point for our subsequences. So, we are going to find the find minimum of some subsequence, and check if that minimum is greater than or equal to the length of that subsequence. For example, our elements are [1,3,5]. We are taking the subsequence (1,2), meaning from the second to the third element. The minimum of that subsequence is 3, which is greater than 2; hence, we can draw there a square. In that fashion we are going to populate our table. As a way to increase performance, we can apply this stratefy: Since in the right upper corner of the table there will be small values, we can use early stopping, when the minimum element for a diagonal is less than the lenght of a subsequence (I know I may have explained it in a vague way, but try to read the code, please). Below there are all the versions of the algorithm, from the slowest to the fastest.

[67]:
```
l = list(range(1000))
```

[68]:
```
def solution_0(A):
    # write your code in Python 3.6
    n = len(A)
    m = [[0 for i in range(n)] for j in range(n)]
    for i in range(0, n):
        m[i][i] = A[i]

    coord = ((i, i + d) for d in range(1, n) for i in range(0, n - d))
    max_ = 1
    for k in coord:
        i = k[0]
        j = k[1]
        d = j - i + 1
        m[i][j] = min(m[i][j - 1], m[i + 1][j])
        min_ = min(m[i][j], d)
        if min_ > max_:
            max_ = min_
    return max_
    pass

%timeit solution_0(l)
```

```
380 ms ± 8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[69]:
```
def solution_1(A):
    # write your code in Python 3.6
    n = len(A)
    m = [[0 for I in range(n)] for j in range(n)]

    for i in range(n):
        m[i][i] = A[i]

    for d in range(1, n):
        for i in range(0, n - d):
            j = i + d
            m[i][j] = min(m[i][j - 1], m[i + 1][j])

    max_ = 1
    for d in range(1, n):
        for i in range(0, n - d):
            j = i + d
            min_ = min(m[i][j], d + 1)
            m[i][j] = min_
            if min_ > max_:
                max_ = min_
    return max_
    pass

print(solution_1(l))
%timeit solution_1(l)
```

```
500
337 ms ± 8.25 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[70]:
```
def solution_2(A):

    n = len(A)
    m = [[0 for I in range(n)] for j in range(n)]

    for i in range(n):
        m[i][i] = A[i]

    max_ = 1
    for d in range(1, n):
        max_for_d = 1

        for i in range(0, n - d):
            j = i + d
            pivot = min(m[i][j - 1], m[i + 1][j])
            m[i][j] = pivot
            min_ = min(pivot, d + 1)
```

```
            if min_ > max_:
                max_ = min_

            if pivot > d:
                max_for_d = pivot

        if max_for_d < d + 1:
            break

    return max_
    pass

print(solution_2(l))
%timeit solution_2(l)
```

```
500
236 ms ± 7.05 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[71]:
```
import numpy as np
from numba import jit

l = np.array(range(1000))
@jit(nopython=True)
def solution_numpy_numba(A):
    n = len(A)
    m = np.zeros((n,n))
    np.fill_diagonal(m, A)
    max_ = 1
    for d in range(1, n):
        max_for_d = 1

        for i in range(0, n - d):
            j = i + d
            pivot = min(m[i, j - 1], m[i + 1, j])
            m[i, j] = pivot
            min_ = min(pivot, d + 1)
            if min_ > max_:
                max_ = min_

            if pivot > d:
                max_for_d = pivot

        if max_for_d < d + 1:
            break

    return max_
    pass
```

```
print(solution_numpy_numba(l))
%timeit solution_numpy_numba(l)
```

```
500.0
2.08 ms ± 41.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[72]:
```
import time
# DO NOT REPORT THIS... COMPILATION TIME IS INCLUDED IN THE EXECUTION TIME!
start = time.time()
solution_numpy_numba(l)
end = time.time()
print("Elapsed (with compilation) = %s" % (end - start))

# NOW THE FUNCTION IS COMPILED, RE-TIME IT EXECUTING FROM CACHE
start = time.time()
solution_numpy_numba(l)
end = time.time()
print("Elapsed (after compilation) = %s" % (end - start))
```

```
Elapsed (with compilation) = 0.002312183380126953
Elapsed (after compilation) = 0.0022041797637939453
```