Algorytmy macierzowe - zadanie nr 2 - Eliminacja Gaussa i LU Faktoryzacja dla macierzy gęstych

"Proszę wybrać język programowania wedle uznania. Proszę napisać procedurę [S]=Schur_Complement(A,n,m) gdzie A to macierz wejściowa, n to rozmiar tej macierzy A, m to rozmiar podmacierzy (tzw. dopełnienia Schura), powstałej poprzez wyeliminowanie n-m wierszy i kolumn z macierzy A, wykorzystując [zatrzymując po n-m krokach]:

6. Faktoryzacja Cholesky'ego "wektorową" (slajd 28) "

Marcin Hawryluk, Norbert Wolniak
grupa: piątek 12:50B

In [1]:
```
import numpy as np
from time import time
import matplotlib.pyplot as plt
import pandas as pd
import os
```

Do implementacji wybraliśmy język Python 3 wraz z biblioteką do obliczeń numerycznych numpy, która pozwala operować na macierzach zaimplementowanych bezpośrednio w języku C.

Generowanie macierzy

Macierze, których będziemy używać, wygenerowaliśmy za pomocą dostarczonej procedury massmatrix, napisanej w środowisku Octave. Macierze zapisaliśmy w postaci pliku tekstowego, a następnie odczytaliśmy w Pythonie za pomocą poniższej funkcji.

In [2]:
```
def read_matrix(file_name):
    with open(file_name, 'r') as file:
        for line in file:
            if line.strip() == '':
                continue
            if line[0] == '#':
                if line[2:6] == "rows":
                    _, _, size = line.split()
                    size = int(size)
                    matrix = np.zeros((size, size))
            else:
                row, col, val = line.split(' ')
                matrix[int(row)-1, int(col)-1] = val

    return matrix
```

In [3]:
```
matrix_small = read_matrix('matrices/riga_2.txt')
```

- przygotowanie zestawu macierzy do testów:

In [4]:
```
fem_matrices  = []
riga_matrices = []

for file in os.listdir('matrices'):
    if file.startswith('riga'):
        riga_matrices.append(read_matrix('matrices/' + file))
    if file.startswith('fem'):
        fem_matrices.append(read_matrix('matrices/' + file))
```

Faktoryzacja Cholesky'ego

Faktoryzacja Cholesky'ego jest procedurą rozkładu macierzy na czynniki L i L.T (transpozycję L) lub w innym wariancie na macierze L, D, L.T, gdzie macierz L to macierz trójkątna dolna, a D macierz przekątniowa; iloczyn tychże dwóch/trzech macierzy dawać ma wyjściową macierz. Faktoryzacja ta jest możliwa jedynie dla macierzy symetrycznych, dodatnio określonych.

- $A = LDL^T$

In [5]:
```
def cholesky_LDLT(matrix):
    A = matrix.copy()
    n = A.shape[0]

    for k in range(n):
        dkk = A[k, k]
        if abs(dkk) < 1e-8:
            raise ValueError('singular matrix')

        vk = A[k+1:n, k].copy()
        A[k+1:n, k] /= dkk

        for j in range(k + 1, n):
            A[j:n, j] -= A[j:n, k] * vk[j-k-1]

    D = np.diag(A)*np.eye(n)

    return np.tril(A) - D + np.eye(n), D
```

- $A = LL^T$

In [6]:
```
def cholesky_LLT(matrix):
    A = matrix.copy()
    n = A.shape[0]

    for k in range(n):
        if abs(A[k, k]) < 1e-8:
            raise ValueError('singular matrix')

        vk = A[k+1:n, k]
        A[k, k] **= 0.5
        dkk = A[k, k]
        A[k+1:n, k] /= dkk

        for j in range(k+1, n):
            A[j:n, j] -= A[j:n, k]*vk[j-k-1]

    return np.tril(A)
```

- testy poprawności:

In [7]:
```
matrix = np.array([[4, 4, 6], [4, 13, 15], [6, 15, 43]], dtype=float)
matrix
```

Out[7]:
```
array([[ 4.,  4.,  6.],
       [ 4., 13., 15.],
       [ 6., 15., 43.]])
```

In [8]:
```
L = cholesky_LLT(matrix)

print("---L L.T---")

print("L:\n", L)
print("\nL.T:\n", L.T)
print("\nL*L.T:\n", L @ L.T)

print("\nnumpy L:\n")
print("", np.linalg.cholesky(matrix), "\n")

print("Correct!" if np.allclose(L @ L.T, matrix) else "wrong")
```

```
---L L.T---
L:
 [[2. 0. 0.]
 [2. 3. 0.]
 [3. 3. 5.]]

L.T:
 [[2. 2. 3.]
 [0. 3. 3.]
 [0. 0. 5.]]

L*L.T:
 [[ 4.  4.  6.]
 [ 4. 13. 15.]
 [ 6. 15. 43.]]

numpy L:

 [[2. 0. 0.]
 [2. 3. 0.]
 [3. 3. 5.]]

Correct!
```

In [9]:
```
L, D = cholesky_LDLT(matrix)

print("---L D L.T---")

print("L:\n", L)
print("\nD:\n", D)
print("\nL.T:\n", L.T)
print("\nL*D*L.T:\n", L @ D @ L.T)
print()
print("Correct!" if np.allclose(L @ D @ L.T, matrix) else "wrong")
```

```
---L D L.T---
L:
 [[1.  0.  0. ]
 [1.  1.  0. ]
 [1.5 1.  1. ]]

D:
 [[ 4.  0.  0.]
 [ 0.  9.  0.]
 [ 0.  0. 25.]]

L.T:
 [[1.  1.  1.5]
 [0.  1.  1. ]
 [0.  0.  1. ]]

L*D*L.T:
 [[ 4.  4.  6.]
 [ 4. 13. 15.]
 [ 6. 15. 43.]]

Correct!
```

Obydwie wersje znajdują poprawne faktoryzacje zadanej macierzy.

Dopełnienie Schura

Do znalezienia uzupełnienia Schura o danym rozmiarze m, wykorzystamy procedurę eliminacji w ramach faktoryzacji Cholesky'ego, zatrzymaną po n-m krokach, zwracając jeszcze nie w pełni przetworzoną podmacierz m x m.

- $A = LDL^T$

In [10]:
```
def schur_LDLT(matrix, m=1):
    A = matrix.copy()
    n = A.shape[0]

    for k in range(n-m):
        dkk = A[k, k]
        if abs(dkk) < 1e-8:
            raise ValueError('singular matrix')

        vk = A[k+1:n, k].copy()
        A[k+1:n, k] /= dkk

        for j in range(k+1, n):
            A[j:n, j] -= A[j:n, k] * vk[j-k-1]

    return A[max(0,n-m):n, max(0,n-m):n]
```

- $A = LL^T$

In [11]:
```
def schur_LLT(matrix, m=1):
    A = matrix.copy()
    n = A.shape[0]

    for k in range(n-m):
        if abs(A[k, k]) < 1e-8:
            raise ValueError('singular matrix')

        vk = A[k+1:n, k]
        A[k, k] **= 0.5
        dkk = A[k, k]
        A[k+1:n, k] /= dkk

        for j in range(k+1, n):
            A[j:n, j] -= A[j:n, k]*vk[j-k-1]

    return A[max(0,n-m):n, max(0,n-m):n]
```

In [12]:
```
print(schur_LDLT(matrix, 2))
print(schur_LLT(matrix, 2))
```

```
[[ 9. 15.]
 [ 9. 34.]]
[[ 9. 15.]
 [ 9. 34.]]
```

Pomiar czasów

"(...) proszę narysować następujący wykres:

> - oś pozioma: rozmiar macierzy n dla liczby przedziałów nxx=2,3,4,… (tak duże macierze ile się uda policzyć na laptopie),
> - oś pionowa: czas [s] (Octave tic; Schur Complement(...); toc)

Proszę narysować

> a) wykres czasu obliczeń dopełnień Schura o rozmiarze n/2
> b) wykres czasu obliczeń dopełnień Schura o rozmiarze n/4
> c) … takie podziały jakie mają sens do rozmiaru 1"

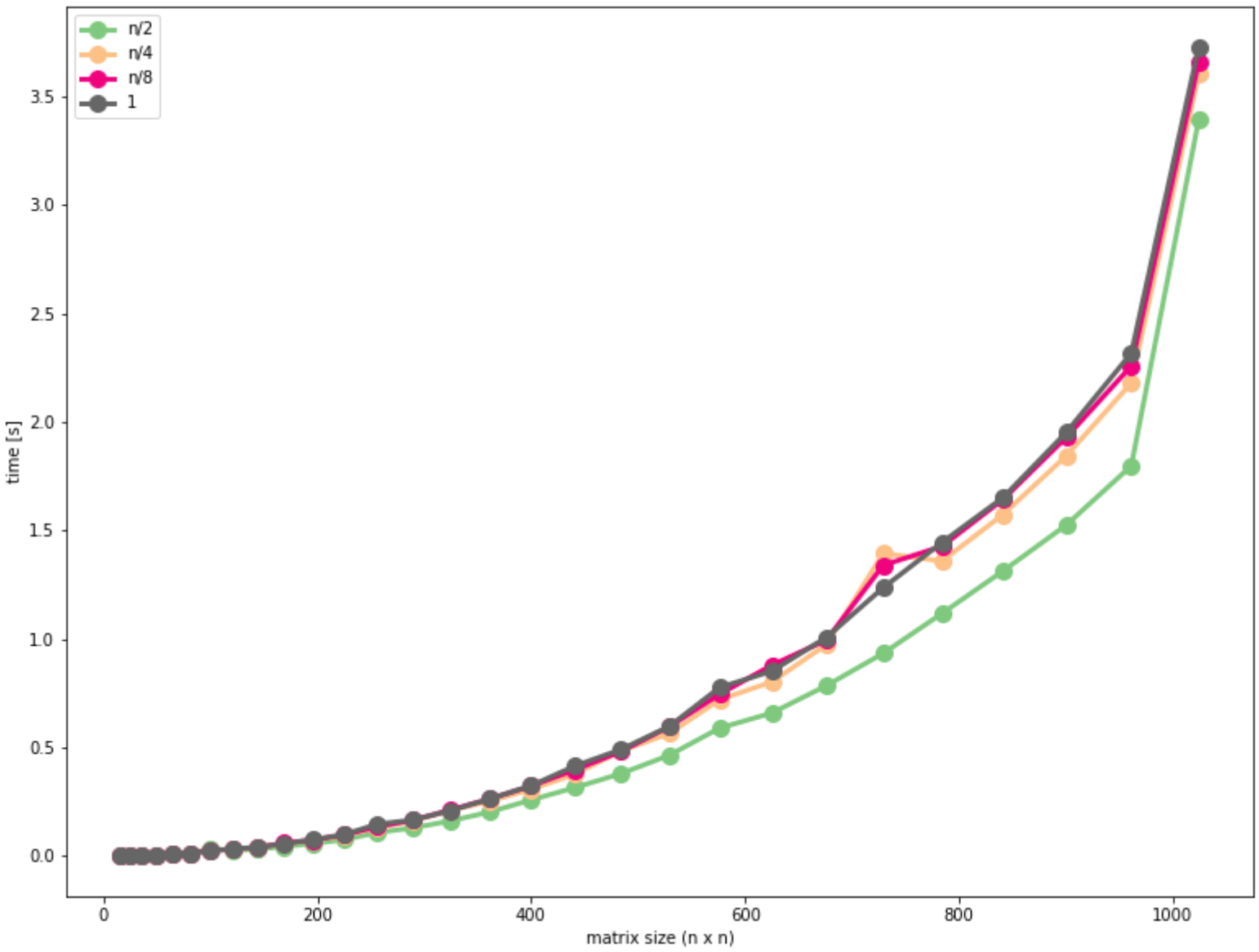
In [13]:
```
def compare_times(matrices):
    times = {}

    for matrix in matrices:
        n = matrix.shape[0]
        times[n] = {}

        start = time()
        schur_LDLT(matrix, n//2)
        times[n]['n/2'] = time() - start

        start = time()
        schur_LDLT(matrix, n//4)
        times[n]['n/4'] = time() - start

        start = time()
        schur_LDLT(matrix, n//8)
        times[n]['n/8'] = time() - start

        start = time()
        schur_LDLT(matrix, 1)
        times[n]['1'] = time() - start

    df = pd.DataFrame(times).T.sort_index()
    df.plot(
        figsize=(13, 10),
        xlabel='matrix size (n x n)',
        ylabel='time [s]',
        colormap='Accent',
        linewidth=3,
        markersize=10,
        marker='o'
    )

    return df
```

- dla macierzy IGA riga=0, pxx=2, rxx=0

In [14]:
```
compare_times(riga_matrices)
```

Out[14]:

	n/2	n/4	n/8	1
16	0.000525	0.000626	0.000508	0.000513
25	0.000913	0.001123	0.001176	0.001241
36	0.002174	0.002734	0.002908	0.002955
49	0.003447	0.004265	0.004387	0.004534
64	0.005663	0.007140	0.007635	0.008114
81	0.009209	0.011464	0.012029	0.012302
100	0.033448	0.023649	0.022369	0.021951
121	0.023523	0.028496	0.030623	0.031748
144	0.029737	0.037118	0.038261	0.038929
169	0.041957	0.055481	0.058461	0.055084
196	0.056350	0.070889	0.072615	0.073646
225	0.074214	0.091380	0.096028	0.097781
256	0.106496	0.129554	0.134220	0.145252
289	0.128119	0.159502	0.163325	0.164364
324	0.159897	0.208101	0.210144	0.207447
361	0.201414	0.250979	0.261877	0.261328
400	0.257296	0.306202	0.321662	0.323191
441	0.314172	0.376534	0.396670	0.413889
484	0.378597	0.482837	0.480679	0.489903
529	0.463105	0.563175	0.594860	0.595943
576	0.588040	0.718833	0.744223	0.775034
625	0.658161	0.801551	0.878330	0.851100
676	0.784124	0.973532	0.994261	1.006731
729	0.932692	1.393625	1.338786	1.235057
784	1.116747	1.358039	1.426206	1.443936
841	1.312185	1.572479	1.642221	1.650312
900	1.523418	1.842370	1.927590	1.950084
961	1.796932	2.178906	2.257127	2.316508
1024	3.391579	3.601968	3.656618	3.726101

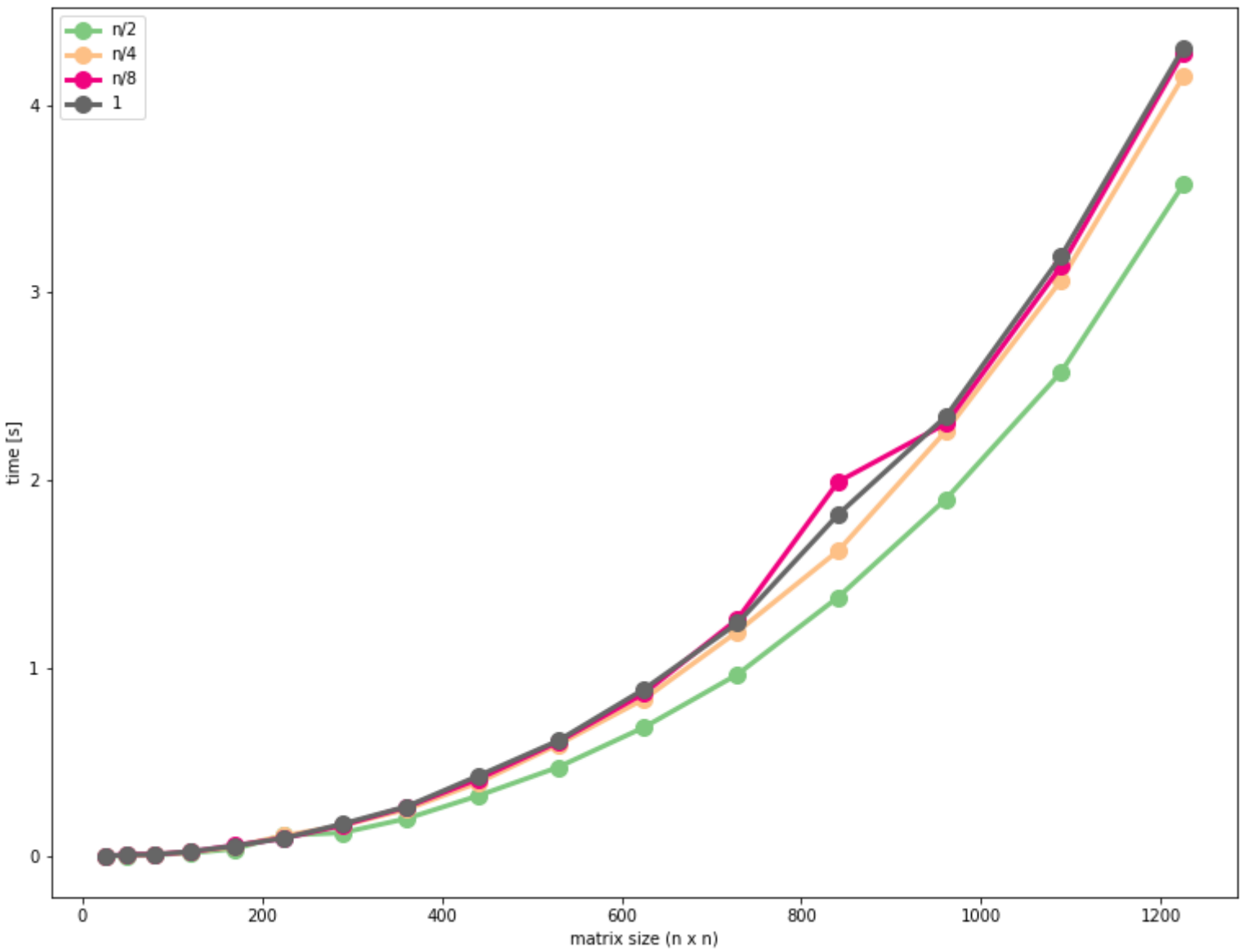


- dla macierzy FEM riga=1, pxx=2, rxx=0

In [15]:
```
compare_times(fem_matrices)
```

Out[15]:

	n/2	n/4	n/8	1
25	0.002499	0.002044	0.001630	0.001634
49	0.004461	0.009955	0.012859	0.011925
81	0.009958	0.012386	0.012645	0.012055
121	0.020535	0.025536	0.026899	0.028393
169	0.041031	0.052924	0.059986	0.057300
225	0.114696	0.113004	0.098655	0.099365
289	0.127268	0.165064	0.167014	0.174114
361	0.203008	0.252755	0.264181	0.267429
441	0.325209	0.395158	0.414411	0.431171
529	0.475407	0.595719	0.609431	0.616011
625	0.688454	0.838776	0.869307	0.891169
729	0.970115	1.198041	1.263888	1.242451
841	1.380602	1.627477	1.994608	1.820465
961	1.902927	2.265076	2.304225	2.342367
1089	2.577405	3.065754	3.140907	3.193733
1225	3.575630	4.149388	4.275998	4.301069



Otrzymane rezultaty przypominają wykres funkcji wielomianowej (sześciennej), co pokrywa się z oczekiwaniami. Otrzymanie dopełnienia o mniejszym rozmiarze zajmuje więcej czasu, gdyż wymaga większej liczby iteracji w procesie eliminacji. Wyniki czasowe są porównywalne dla dwóch

typów macierzy, nie zaobserwowaliśmy wyraźnej przewagi którejkolwiek z nich.

Koszt obliczeniowy i pamięciowy

"3. Jaki jest koszt obliczeniowy i pamięciowy (flopsy i memopsy) zaimplementowanego algorytmu?"

Poniższe obliczenia dotyczą faktoryzacji Cholesky'ego w wersji LDL.T:

Koszt obliczeniowy (flops):

$$\sum_{k=0}^{n-1}[(n-k-1)+\sum_{j=k+1}^{n-1}(n-j)*2]$$

$$\sum_{k=0}^{n-1}[(n-k-1)+(k-n)*(1+k-n)]$$

$$\frac{1}{3}n^3+\frac{1}{2}n^2-\frac{5}{6}n$$

Koszt pamięciowy (memops):

$$\sum^{n-1}[3+4*(n-k-1)+\sum^{n-1}(1+2*(n-j))]$$

$$\sum_{k=0}^{n-1}[3+4*(n-k-1)+(k-n)^2-1]$$

$$\frac{1}{3}n^3+\frac{15}{6}n^2+\frac{1}{6}n$$

Koszt procedury obliczania dopełnienia Schura znajdujemy analogicznie, jedynie sumując do k=n-m-1. Koszt zależy wtedy od wybranego rozmiaru m podmacierzy.

Wnioski

- Za pomocą zarówno eliminacji Gaussa, jak i faktoryzacji Cholesky'ego, jesteśmy w stanie otrzymać dopełnienie Schura, które znajduje zastosowanie w wielu działach matematyki, m.in. w statystyce, rachunku prawdopodobieństwa, algebrze (do rozwiązywania układu równań liniowych).
- Algorytm faktoryzacji LDL^T ma porównywalne koszty obliczeniowe i pamięciowe rzędu n^3.

M. Hawryluk, N. Wolniak. 2021